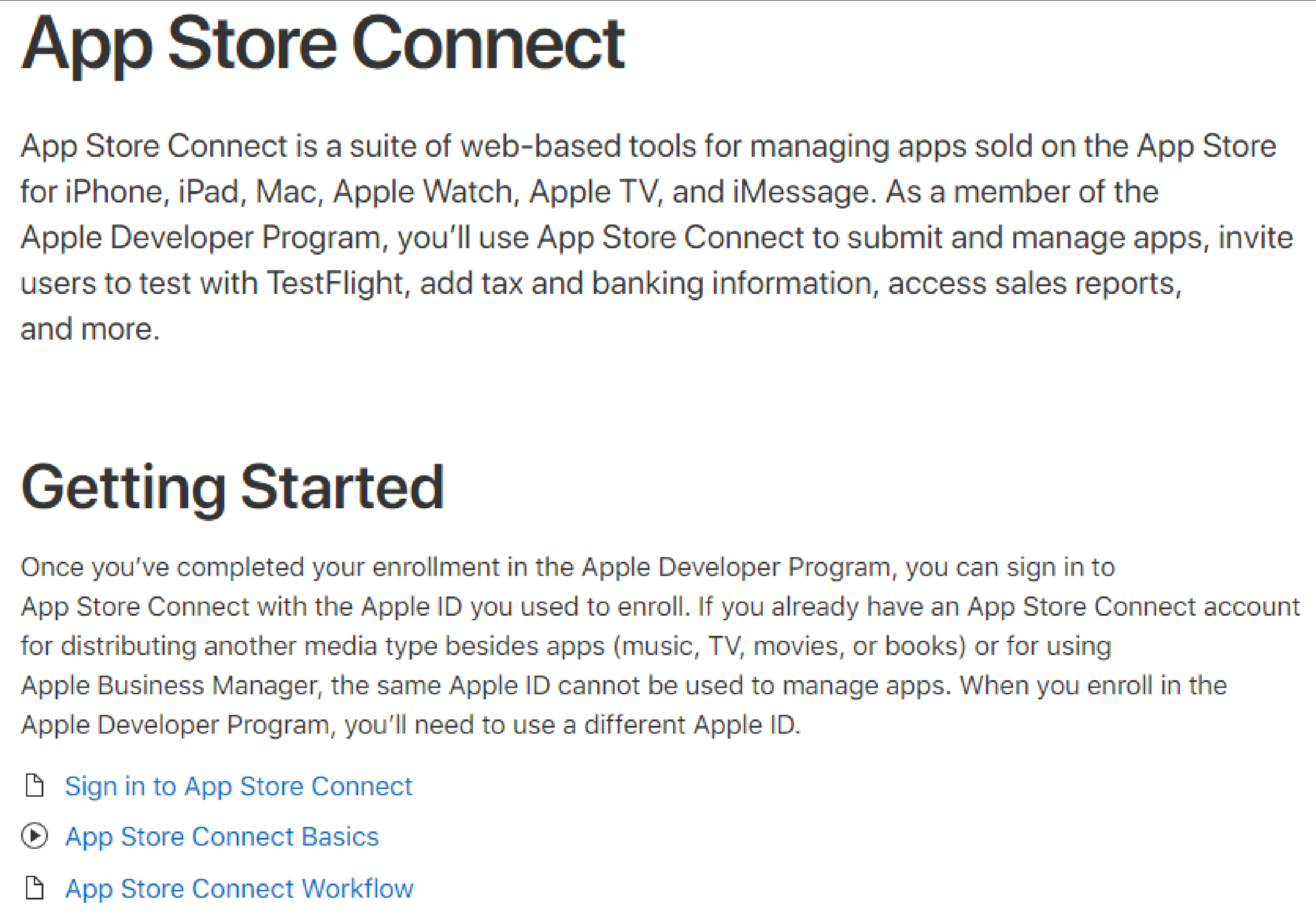
Exhibit 2b

Infringement Contentions – Asana's iOS Mobile Apps

U.S. Patent No. 5,991,399

Claim	Analysis
1. A method of securely distributing data comprising:	Asana's mobile software application products and services including by way of example, but not limited to the following apps ("mobile applications", "mobile apps" or "Accused Products") that are specifically developed, used, sold, offered for sale, marketed, licensed and distributed by Asana to be downloaded onto Apple iOS mobile or tablet devices. • Asana: organize tasks & work (https://apps.apple.com/us/app/asana-organize-tasks-work/id489969512), Last accessed on Mar 19, 2020 Asana directly infringes and/or continues to knowingly induce Apple to infringe this claim by intentionally developing, making, marketing, advertising, providing, sending, distributing and licensing its mobile applications software, documentation, materials, training or support and aiding, abetting, encouraging, promoting or inviting use thereof. To the extent any steps identified herein are performed by Apple, such acts are attributable to Asana (i) because Asana works together with Apple in a joint enterprise in the building and distribution of its mobile apps, or (ii) because Apple distributes and markets Asana's mobile apps under the direction and control of Asana, or acts as agent, or on behalf of Asana, in the building, marketing and distribution of Asana's mobile apps. Alternatively, any steps or acts performed by Asana, are attributable to Apple, who conditions participation in and the receipt of a benefit, namely, the distribution of Asana's mobile apps through its app store, upon compliance with certain mandatory procedures and guidelines dictated by Apple in the building and upload of Asana's mobile apps, and Asana induces infringement by Apple in the building, marketing and distribution of Asana's mobile apps. To the extent the preamble is limiting, Asana distributes data according to the method of claim 1 as set forth below. In order to build and send the mobile app securely to the Apple servers, Defendant practices the method of claim 1 as set forth below in order to securely distribute its mobile app to Defendant's customers through the Apple App Store. As described below, Defendant registers with the App Store Connect portal (https://appstoreconnect.apple.com, Last accessed on Mar 19, 2020) in order to securely upload Asana's apps onto the Apple App Store, as evidenced by the "https" in its URL.



App Store Connect

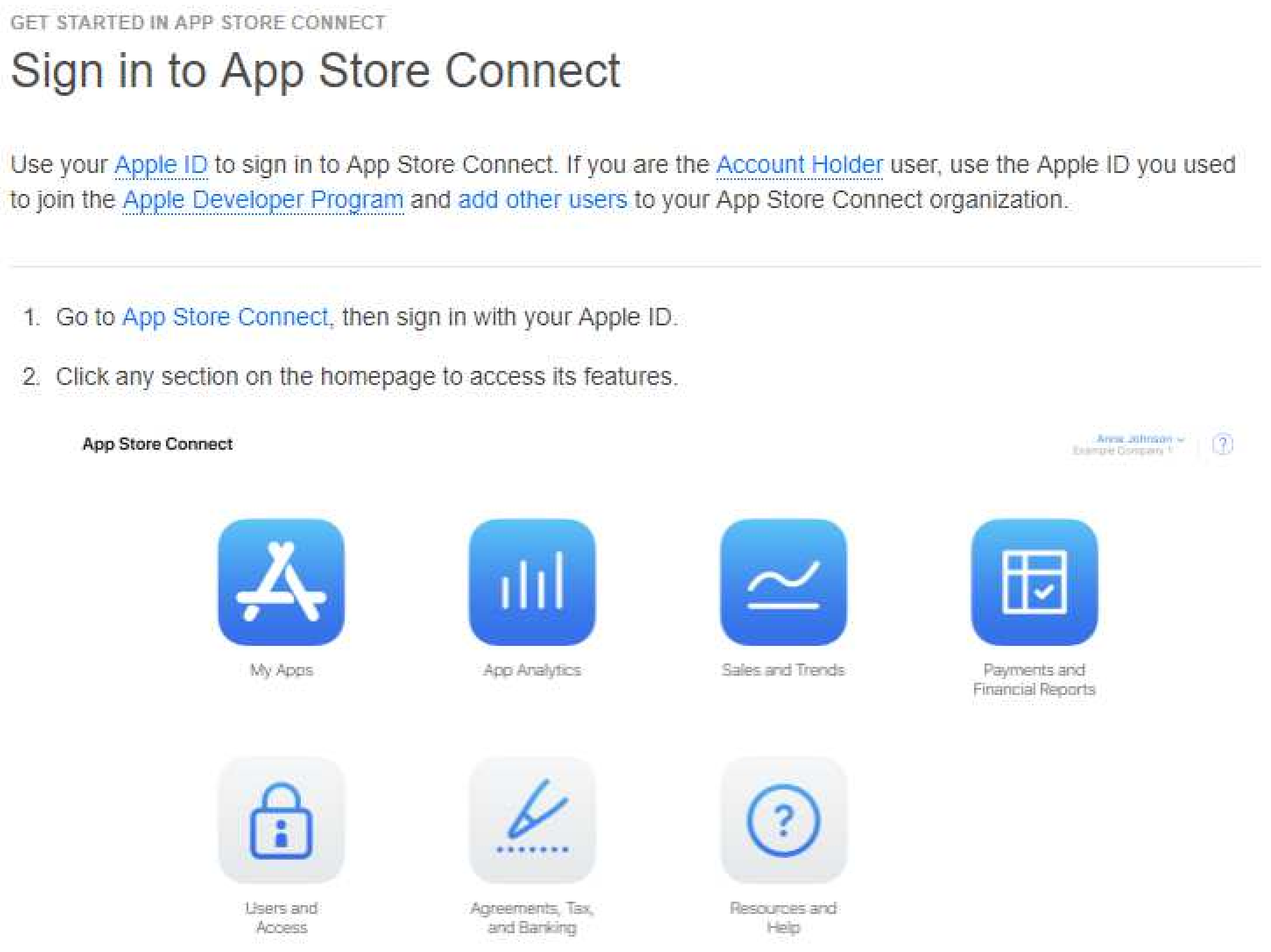
App Store Connect is a suite of web-based tools for managing apps sold on the App Store for iPhone, iPad, Mac, Apple Watch, Apple TV, and iMessage. As a member of the Apple Developer Program, you'll use App Store Connect to submit and manage apps, invite users to test with TestFlight, add tax and banking information, access sales reports, and more.

Getting Started

Once you've completed your enrollment in the Apple Developer Program, you can sign in to App Store Connect with the Apple ID you used to enroll. If you already have an App Store Connect account for distributing another media type besides apps (music, TV, movies, or books) or for using Apple Business Manager, the same Apple ID cannot be used to manage apps. When you enroll in the Apple Developer Program, you'll need to use a different Apple ID.

- Sign in to App Store Connect
- App Store Connect Basics
- App Store Connect Workflow

Source: https://developer.apple.com/support/app-store-connect/, Last accessed on Mar 19, 2020



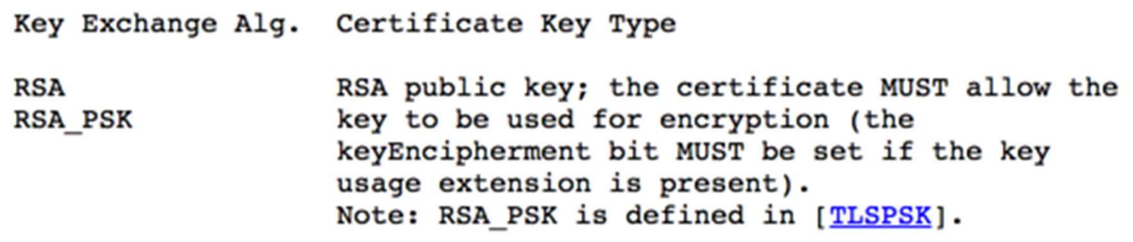
GET STARTED IN APP STORE CONNECT

Sign in to App Store Connect

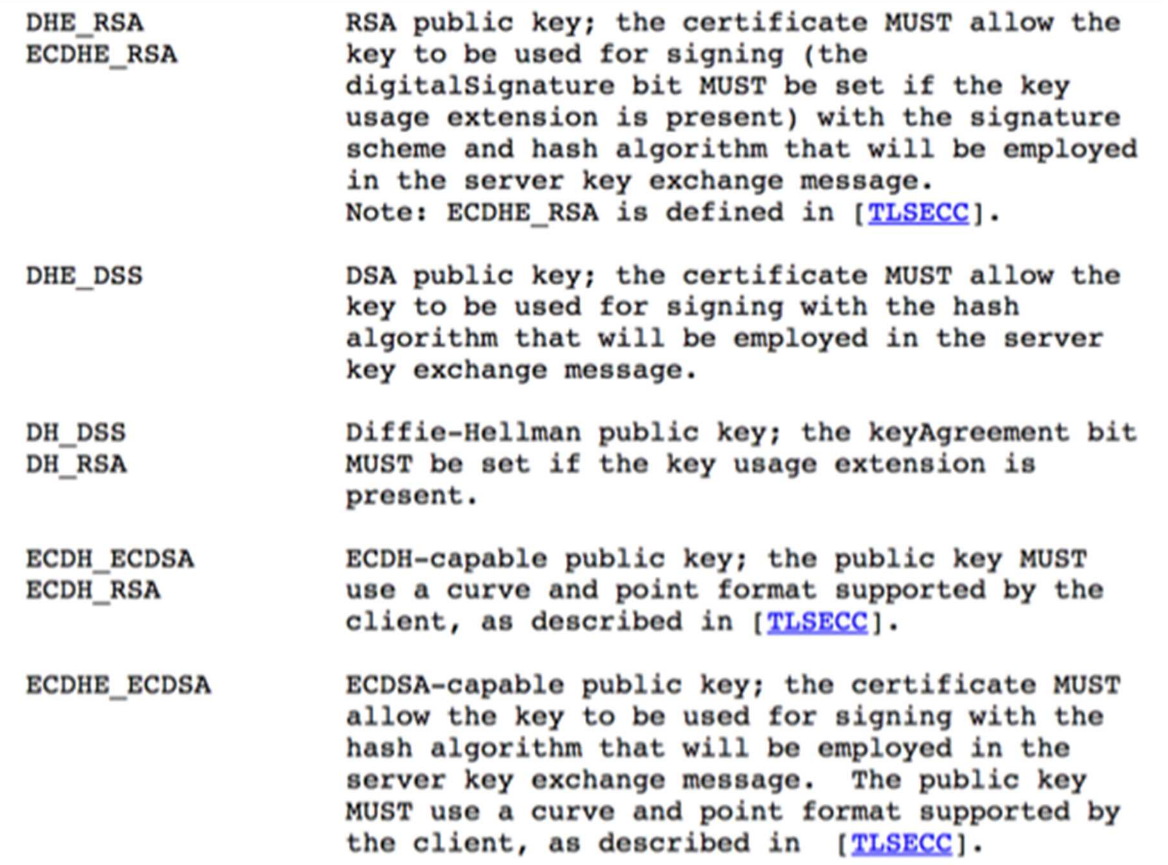
Use your Apple ID to sign in to App Store Connect. If you are the Account Holder user, use the Apple ID you used to join the Apple Developer Program and add other users to your App Store Connect organization.

1. Go to App Store Connect, then sign in with your Apple ID.
2. Click any section on the homepage to access its features.

Source: https://help.apple.com/app-store-connect/#/devcd5016d31, Last accessed on Mar 19, 2020

generating an asymmetric key pair having a public key and a private key;	The Court previously construed[1] "an asymmetric key pair having a public key and a private key" in claim 1 to mean "one or more asymmetric key pairs, one of the asymmetric key pairs having the claimed public key and claimed private key, the asymmetric keys of an asymmetric key pair being complementary by performing complementary functions, such as encrypting and decrypting data or creating and verifying signatures."
	
	Also relevant to this claim element is the Court's previous construction of "executable tamper resistant key module" / "executable tamper resistant code module" / "tamper resistant key module" to mean "software that is designed to work with other software, that is resistant to observation and modification, and that includes a key for secure communication."
	
	Also relevant to this claim element is the Court's rejection of limiting "including" to compiling.
	
	Upon information and belief, the method step of "generating an asymmetric key pair having a public key and a private key" is performed by Apple and/or its agents – whose acts are attributable to Asana (i) because Asana works together with Apple in a joint enterprise in the building and distribution of its mobile apps, or (ii) because Apple distributes and markets Asana's mobile apps under the direction and control of Asana, or acts as agent, or on behalf of Asana, in the building, marketing and distribution of Asana's mobile apps.
	
	Asana uploads their mobile apps to Apple using a TLS connection – which begins with a TLS handshake. A TLS handshake is a mandatory procedure that allows Asana and Apple to exchange cryptographic parameters, including a cipher suite and arrive at a shared master secret for encrypting communication including upload of Asana's mobile apps to Apple servers.
	
	A TLS handshake begins with Asana sending a list of cipher suites supported by Asana to Apple. These cipher suites specify at least one or more of the following key exchange algorithms:
	
	
	`Key Exchange Alg.  Certificate Key Type`
	
	`RSA                RSA public key; the certificate MUST allow the`
	`RSA_PSK            key to be used for encryption (the`
	`                   keyEncipherment bit MUST be set if the key`
	`                   usage extension is present).`
	`                   Note: RSA_PSK is defined in [TLSPSK].`

[1] See Memorandum Opinion and Order, Document 104 signed by Judge Rodney Gilstrap on 7/21/2016 in re Plano Encryption Technologies, LLC v. American Bank of Texas (2:15-cv-01273).



DHE_RSA ECDHE_RSA	RSA public key; the certificate MUST allow the key to be used for signing (the digitalSignature bit MUST be set if the key usage extension is present) with the signature scheme and hash algorithm that will be employed in the server key exchange message. Note: ECDHE_RSA is defined in [TLSECC].
DHE_DSS	DSA public key; the certificate MUST allow the key to be used for signing with the hash algorithm that will be employed in the server key exchange message.
DH_DSS DH_RSA	Diffie-Hellman public key; the keyAgreement bit MUST be set if the key usage extension is present.
ECDH_ECDSA ECDH_RSA	ECDH-capable public key; the public key MUST use a curve and point format supported by the client, as described in [TLSECC].
ECDHE_ECDSA	ECDSA-capable public key; the certificate MUST allow the key to be used for signing with the hash algorithm that will be employed in the server key exchange message. The public key MUST use a curve and point format supported by the client, as described in [TLSECC].

Source: https://tools.ietf.org/html/rfc5246 at 48-49, Last accessed on Mar 19, 2020

Each of these algorithms necessitates generating one or more asymmetric key pairs.

For **RSA and RSA _PSK**, an Apple server generates an RSA public-private key pair. The RSA public key and the RSA private key are complementary, by performing complementary function encrypting and decrypting data.

	For **DHE_RSA, ECDHE_RSA, DH_RSA and ECDH_RSA**, Apple server generates an RSA public-private key pair as well as a Diffie-Hellman ("DH") public-private key pair[2]. Asana also generates a second Diffie-Hellman public-private key pair. The RSA public key and the RSA private key are complementary by performing complementary functions, such as creating and verifying signatures. The Diffie-Hellman public key and the Diffie-Hellman private key are also complementary by performing complementary functions, such as encrypting and decrypting data. Specifically, as per the Diffie-Hellman key exchange algorithm, Asana uses Apple's Diffie-Hellman public key combined with Asana's own Diffie-Hellman private key to compute a premaster secret and thereon a master secret[3] for encrypting data. Apple in turn uses Asana's Diffie-Hellman public key combined with Apple's own Diffie-Hellman private key to compute the same premaster secret and the master secret for decrypting encrypted data from Asana. Conversely, Apple uses Asana's Diffie-Hellman public key combined with Apple's own Diffie-Hellman private key to compute a premaster secret and thereon, a master secret[4] for encrypting data. Asana uses Apple's Diffie-Hellman public key combined with Asana's own Diffie-Hellman private key to compute a premaster secret and thereon a master secret for decrypting encrypted data from Apple. For **DHE_DSS and DH_DSS**, Apple server generates a DSA public-private key pair as well as a Diffie-Hellman public-private key pair. Asana also generates a second Diffie-Hellman public-private key pair. The DSA public key and the DSA private key are complementary by performing complementary functions, such as creating and verifying signatures. The Diffie-Hellman public key and the Diffie-Hellman private key are also complementary by performing complementary functions, such as encrypting and decrypting data. Specifically, as per the Diffie-Hellman key exchange algorithm, Asana uses Apple's Diffie-Hellman public key combined with Asana's own Diffie-Hellman private key to compute a premaster secret and thereon a master secret[5] for encrypting data. Apple in turn uses Asana's Diffie-Hellman public key combined with Apple's own Diffie-Hellman private key to compute the same premaster secret and the master secret for decrypting encrypted data from Asana. Conversely, Apple uses Asana's Diffie-Hellman public key combined with Apple's own Diffie-Hellman private key to compute a premaster secret and thereon, a master

[2] ECDH and ECDHE algorithms require generating at the server and the client, elliptical curve parameters that constitute a Diffie-Hellman public-private key pair. See, for example, https://tools.ietf.org/html/rfc5246 page 49-52, https://tools.ietf.org/html/rfc7525 page 12, http://www.cse.hut.fi/fi/opinnot/T-110.5241/2011/luennot-files/Network%20Security%2004%20-%20TLS.pdf, http://csrc.nist.gov/groups/STM/cmvp/documents/140-1/140sp/140sp2897.pdf, http://www.networkworld.com/article/2268575/lan-wan/chapter-2--ssl-vpn-technology.html, http://homes.esat.kuleuven.be/~fvercaut/papers/ACM2012.pdf, Implementing SSL / TLS Using Cryptography and PKI by Joshua Davies, ISBN 1118038770, 9781118038772, Page 305 and Introduction to Computer Networks and Cybersecurity By Chwan-Hwa (John) Wu, J. David Irwin, ISBN 1466572140, 9781466572140, Page 1021, which state that ECDH requires generating a Diffie-Hellman public-private key pair.(Last accessed on Mar 19, 2020)

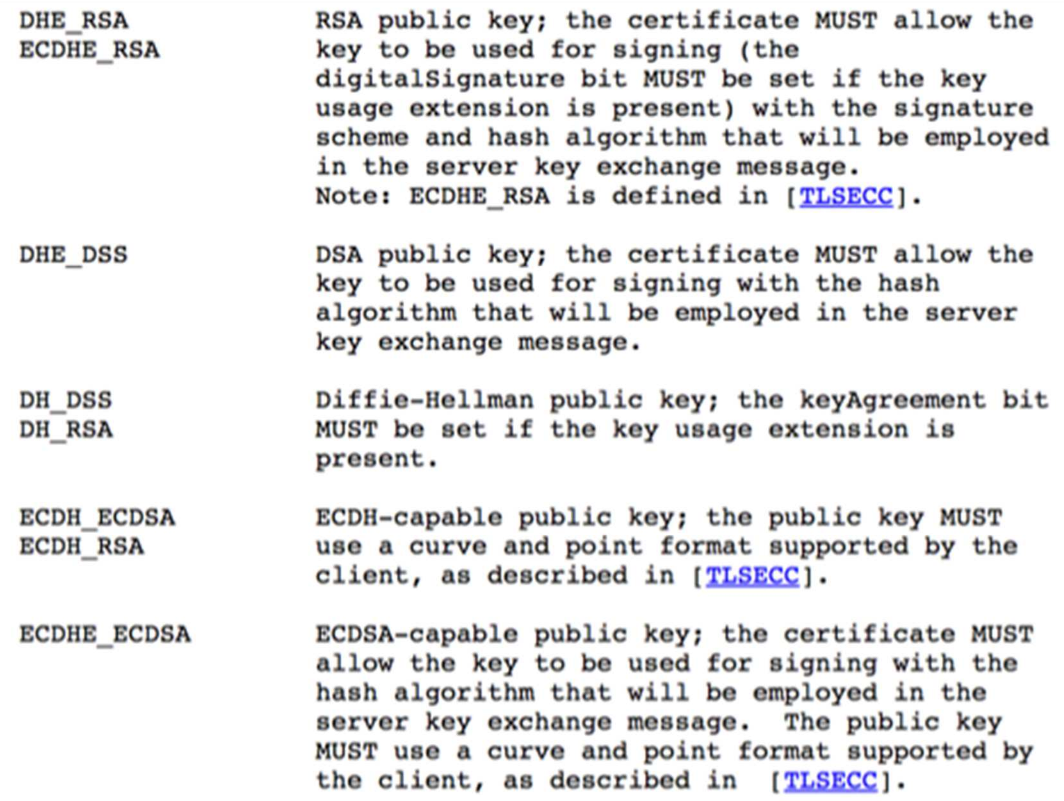
[3] The master secret obtained from the premaster secret may be hashed according to a hashing algorithm also specified in the cipher suite in order to remove weak bits, as explained in https://tools.ietf.org/html/rfc5246, Sections 6.3, 7.4.9 and 8.1.

[4] The master secret obtained from the premaster secret may be hashed according to a hashing algorithm also specified in the cipher suite in order to remove weak bits, as explained in https://tools.ietf.org/html/rfc5246, Sections 6.3, 7.4.9 and 8.1.

[5] The master secret obtained from the premaster secret may be hashed according to a hashing algorithm also specified in the cipher suite in order to remove weak bits, as explained in https://tools.ietf.org/html/rfc5246, Sections 6.3, 7.4.9 and 8.1.

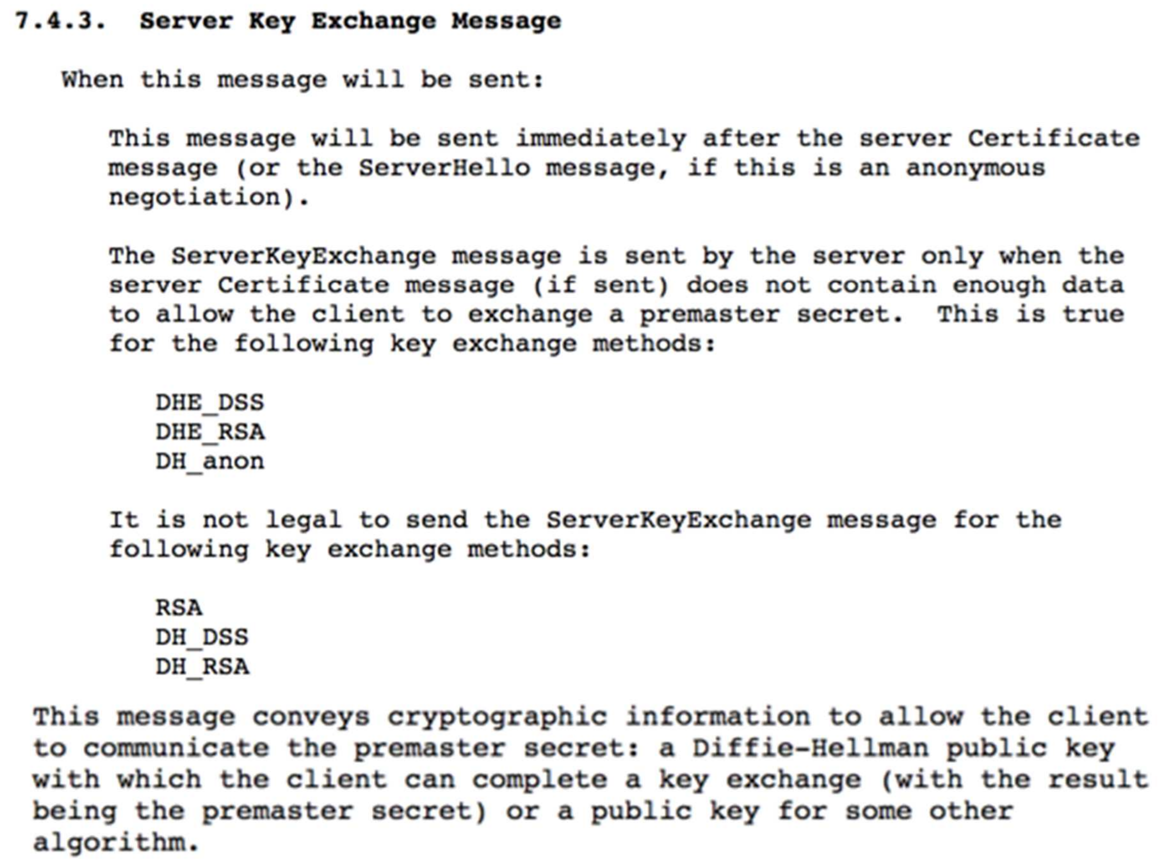
	secret[6] for encrypting data. Asana uses Apple's Diffie-Hellman public key combined with Asana's own Diffie-Hellman private key to compute a premaster secret and thereon a master secret for decrypting encrypted data from Apple. For **ECDH_ECDSA and ECDHE_ECDSA**, an Apple server generates an ECDSA public-private key pair as well as a Diffie-Hellman public-private key pair. Asana also generates a second Diffie-Hellman public-private key pair. The ECDSA public key and the ECDSA private key are complementary by performing complementary functions, such as creating and verifying signatures. The Diffie-Hellman public key and the Diffie-Hellman private key are also complementary by performing complementary functions, such as encrypting and decrypting data. Specifically, as per the Diffie-Hellman key exchange algorithm, Asana uses Apple's Diffie-Hellman public key combined with Asana's own Diffie-Hellman private key to compute a premaster secret and thereon a master secret[7] for encrypting data. Apple in turn uses Asana's Diffie-Hellman public key combined with Apple's own Diffie-Hellman private key to compute the same premaster secret and the master secret for decrypting encrypted data from Asana. Conversely, Apple uses Asana's Diffie-Hellman public key combined with Apple's own Diffie-Hellman private key to compute a premaster secret and thereon, a master secret[8] for encrypting data. Asana uses Apple's Diffie-Hellman public key combined with Asana's own Diffie-Hellman private key to compute a premaster secret and thereon a master secret for decrypting encrypted data from Apple.

[6] The master secret obtained from the premaster secret may be hashed according to a hashing algorithm also specified in the cipher suite in order to remove weak bits, as explained in https://tools.ietf.org/html/rfc5246, Sections 6.3, 7.4.9 and 8.1.
[7] The master secret obtained from the premaster secret may be hashed according to a hashing algorithm also specified in the cipher suite in order to remove weak bits, as explained in https://tools.ietf.org/html/rfc5246, Sections 6.3, 7.4.9 and 8.1.
[8] The master secret obtained from the premaster secret may be hashed according to a hashing algorithm also specified in the cipher suite in order to remove weak bits, as explained in https://tools.ietf.org/html/rfc5246, Sections 6.3, 7.4.9 and 8.1.



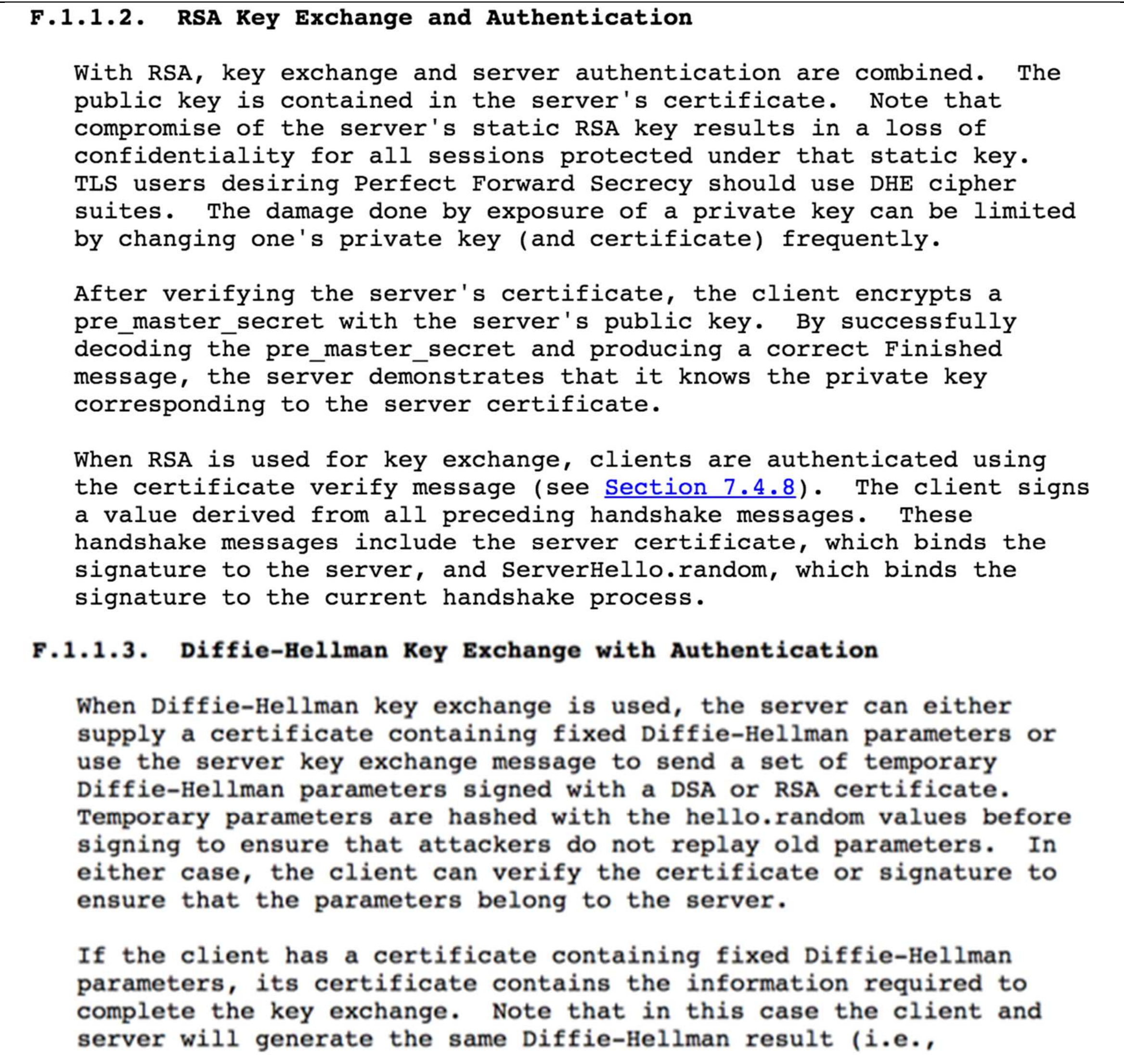
```
   DHE_RSA              RSA public key; the certificate MUST allow the
   ECDHE_RSA            key to be used for signing (the
                        digitalSignature bit MUST be set if the key
                        usage extension is present) with the signature
                        scheme and hash algorithm that will be employed
                        in the server key exchange message.
                        Note: ECDHE_RSA is defined in [TLSECC].

   DHE_DSS              DSA public key; the certificate MUST allow the
                        key to be used for signing with the hash
                        algorithm that will be employed in the server
                        key exchange message.

   DH_DSS               Diffie-Hellman public key; the keyAgreement bit
   DH_RSA               MUST be set if the key usage extension is
                        present.

   ECDH_ECDSA           ECDH-capable public key; the public key MUST
   ECDH_RSA             use a curve and point format supported by the
                        client, as described in [TLSECC].

   ECDHE_ECDSA          ECDSA-capable public key; the certificate MUST
                        allow the key to be used for signing with the
                        hash algorithm that will be employed in the
                        server key exchange message.  The public key
                        MUST use a curve and point format supported by
                        the client, as described in  [TLSECC].
```

Source: https://tools.ietf.org/html/rfc5246 at 48-49, Last accessed on Mar 19, 2020



```
7.4.3.  Server Key Exchange Message

   When this message will be sent:

      This message will be sent immediately after the server Certificate
      message (or the ServerHello message, if this is an anonymous
      negotiation).

      The ServerKeyExchange message is sent by the server only when the
      server Certificate message (if sent) does not contain enough data
      to allow the client to exchange a premaster secret.  This is true
      for the following key exchange methods:

         DHE_DSS
         DHE_RSA
         DH_anon

      It is not legal to send the ServerKeyExchange message for the
      following key exchange methods:

         RSA
         DH_DSS
         DH_RSA

 This message conveys cryptographic information to allow the client
 to communicate the premaster secret: a Diffie-Hellman public key
 with which the client can complete a key exchange (with the result
 being the premaster secret) or a public key for some other
 algorithm.
```

Source: https://tools.ietf.org/html/rfc5246 at 50-51, Last accessed on Mar 19, 2020



F.1.1.2. RSA Key Exchange and Authentication

With RSA, key exchange and server authentication are combined. The public key is contained in the server's certificate. Note that compromise of the server's static RSA key results in a loss of confidentiality for all sessions protected under that static key. TLS users desiring Perfect Forward Secrecy should use DHE cipher suites. The damage done by exposure of a private key can be limited by changing one's private key (and certificate) frequently.

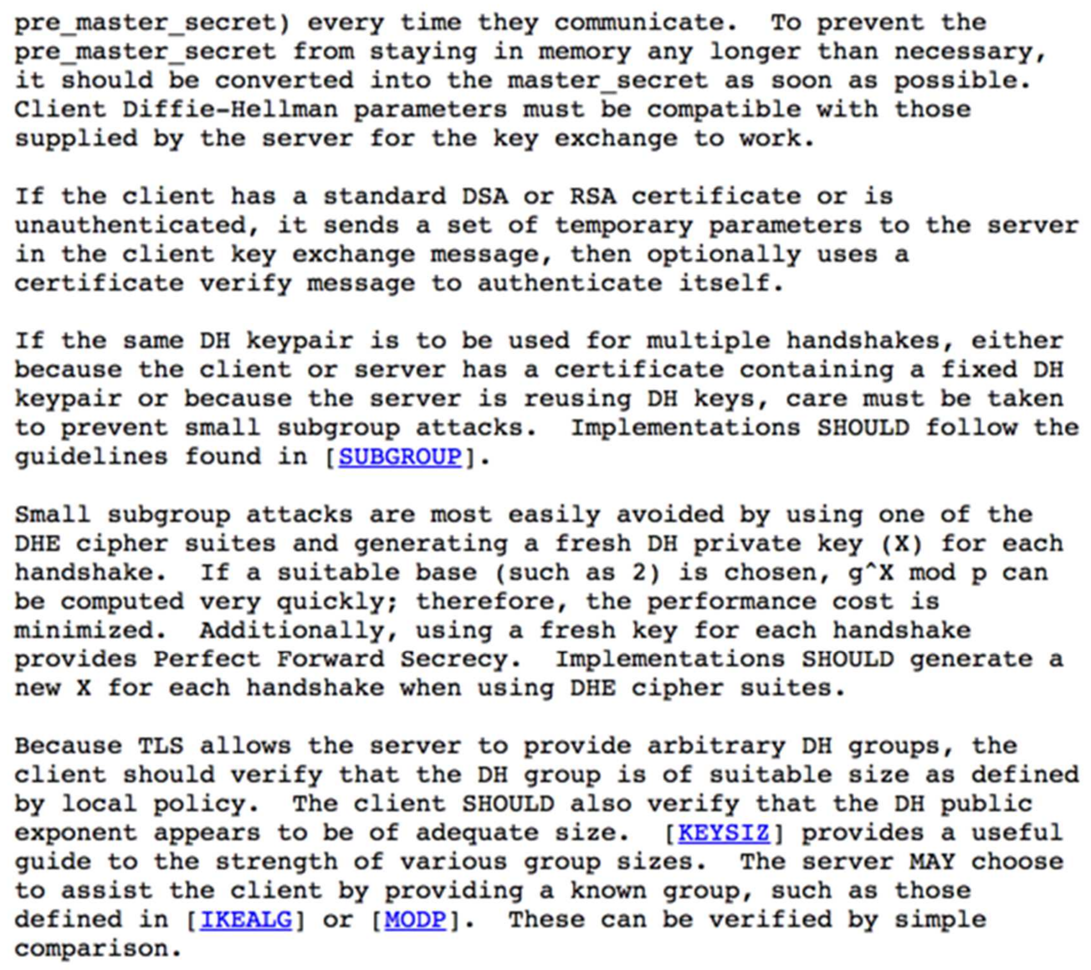
After verifying the server's certificate, the client encrypts a pre_master_secret with the server's public key. By successfully decoding the pre_master_secret and producing a correct Finished message, the server demonstrates that it knows the private key corresponding to the server certificate.

When RSA is used for key exchange, clients are authenticated using the certificate verify message (see Section 7.4.8). The client signs a value derived from all preceding handshake messages. These handshake messages include the server certificate, which binds the signature to the server, and ServerHello.random, which binds the signature to the current handshake process.

F.1.1.3. Diffie-Hellman Key Exchange with Authentication

When Diffie-Hellman key exchange is used, the server can either supply a certificate containing fixed Diffie-Hellman parameters or use the server key exchange message to send a set of temporary Diffie-Hellman parameters signed with a DSA or RSA certificate. Temporary parameters are hashed with the hello.random values before signing to ensure that attackers do not replay old parameters. In either case, the client can verify the certificate or signature to ensure that the parameters belong to the server.

If the client has a certificate containing fixed Diffie-Hellman parameters, its certificate contains the information required to complete the key exchange. Note that in this case the client and server will generate the same Diffie-Hellman result (i.e.,



```
pre_master_secret) every time they communicate.  To prevent the
pre_master_secret from staying in memory any longer than necessary,
it should be converted into the master_secret as soon as possible.
Client Diffie-Hellman parameters must be compatible with those
supplied by the server for the key exchange to work.

If the client has a standard DSA or RSA certificate or is
unauthenticated, it sends a set of temporary parameters to the server
in the client key exchange message, then optionally uses a
certificate verify message to authenticate itself.

If the same DH keypair is to be used for multiple handshakes, either
because the client or server has a certificate containing a fixed DH
keypair or because the server is reusing DH keys, care must be taken
to prevent small subgroup attacks.  Implementations SHOULD follow the
guidelines found in [SUBGROUP].

Small subgroup attacks are most easily avoided by using one of the
DHE cipher suites and generating a fresh DH private key (X) for each
handshake.  If a suitable base (such as 2) is chosen, g^X mod p can
be computed very quickly; therefore, the performance cost is
minimized.  Additionally, using a fresh key for each handshake
provides Perfect Forward Secrecy.  Implementations SHOULD generate a
new X for each handshake when using DHE cipher suites.

Because TLS allows the server to provide arbitrary DH groups, the
client should verify that the DH group is of suitable size as defined
by local policy.  The client SHOULD also verify that the DH public
exponent appears to be of adequate size.  [KEYSIZ] provides a useful
guide to the strength of various group sizes.  The server MAY choose
to assist the client by providing a known group, such as those
defined in [IKEALG] or [MODP].  These can be verified by simple
comparison.
```

Source: The Transport Layer Security (TLS) Protocol Version 1.2, IETF RFC 5246, https://tools.ietf.org/html/rfc5246, Last accessed on Mar 19, 2020

In addition, Asana and/or Apple also generate additional asymmetric key pairs for code-signing the mobile app prior to upload and during TLS communications during the operation of the mobile app.

Thus, at least one asymmetric key pair is generated having the claimed public key and claimed private key in building the tamper resistant app so that the mobile app code and metadata can be sent securely to the Apple servers by SSL/TLS. This asymmetric key pair (as with the asymmetric key pair

	used to digitally sign the app) is complementary as described below by performing complementary functions, such as encrypting and decrypting data and/or creating and verifying signatures. As described in greater detail below, the claimed public and private key are generated and used to securely upload the mobile app onto the Apple servers by SSL/TLS when building the app, where the app includes the generated private key of the claimed asymmetric key pair and the encrypted predetermined data that has been encrypted with the generated public key of the claimed key pair.
encrypting predetermined data with the generated public key;	In order to send the mobile app code securely to the Apple servers, data that is determined prior to encryption including for example, the pre-master secret, is encrypted with the generated public key of the claimed asymmetric key pair, as part of the SSL/TLS process. That process necessarily uses the public key of the generated asymmetric key pair to encrypt data including by way of example, the pre-master secret that is used to create a symmetric key for secure communications. Upon information and belief, the method step of "encrypting predetermined data with the generated public key" is performed by Asana and/or its agents. To the extent any portion of the method step is performed by Apple and/or its agents, such acts are attributable to Asana (i) because Asana works together with Apple in a joint enterprise in the building and distribution of its mobile apps, or (ii) because Apple distributes and markets Asana's mobile apps under the direction and control of Asana, or acts as agent, or on behalf of Asana, in the building, marketing and distribution of Asana's mobile apps. When Asana uploads its mobile apps to Apple, it connects to Apple App Store Connect portal using SSL/TLS protocol. Apple and Asana perform a TLS handshake procedure which uses asymmetric key encryption and necessitates generation of at least one asymmetric key pair. Specifically, Asana negotiates with Apple a cipher suite and the key exchange algorithm that will be used for the handshake. The cryptographic parameters of the session state are produced by the TLS Handshake Protocol, which operates on top of the TLS record layer. When a TLS client and server first start communicating, they agree on a protocol version, select cryptographic algorithms, optionally authenticate each other, and use public-key encryption techniques to generate shared secrets. The TLS Handshake Protocol involves the following steps: - Exchange hello messages to agree on algorithms, exchange random values, and check for session resumption.

- Exchange the necessary cryptographic parameters to allow the client and server to agree on a premaster secret.

- Exchange certificates and cryptographic information to allow the client and server to authenticate themselves.

- Generate a master secret from the premaster secret and exchanged random values.

- Provide security parameters to the record layer.

- Allow the client and server to verify that their peer has calculated the same security parameters and that the handshake occurred without tampering by an attacker.

Source: The Transport Layer Security (TLS) Protocol Version 1.2, IETF RFC 5246, https://tools.ietf.org/html/rfc5246, Last accessed on Mar 19, 2020

The actual key exchange uses up to four messages: the server Certificate, the ServerKeyExchange, the client Certificate, and the ClientKeyExchange. New key exchange methods can be created by specifying a format for these messages and by defining the use of the messages to allow the client and server to agree upon a shared secret. This secret MUST be quite long; currently defined key exchange methods exchange secrets that range from 46 bytes upwards.

Following the hello messages, the server will send its certificate in a Certificate message if it is to be authenticated. Additionally, a ServerKeyExchange message may be sent, if it is required (e.g., if the server has no certificate, or if its certificate is for signing only). If the server is authenticated, it may request a certificate from the client, if that is appropriate to the cipher suite selected. Next, the server will send the ServerHelloDone message, indicating that the hello-message phase of the handshake is complete. The

	server will then wait for a client response. If the server has sent a CertificateRequest message, the client MUST send the Certificate message. The ClientKeyExchange message is now sent, and the content of that message will depend on the public key algorithm selected between the ClientHello and the ServerHello. If the client has sent a certificate with signing ability, a digitally-signed CertificateVerify message is sent to explicitly verify possession of the private key in the certificate. Source: The Transport Layer Security (TLS) Protocol Version 1.2, IETF RFC 5246, https://tools.ietf.org/html/rfc5246, Last accessed on Mar 19, 2020 **The TLS Handshaking Protocols** TLS has three subprotocols that are used to allow peers to agree upon security parameters for the record layer, to authenticate themselves, to instantiate negotiated security parameters, and to report error conditions to each other. The Handshake Protocol is responsible for negotiating a session, which consists of the following items: session identifier An arbitrary byte sequence chosen by the server to identify an active or resumable session state. peer certificate X509v3 [PKIX] certificate of the peer. This element of the state may be null. compression method The algorithm used to compress data prior to encryption. cipher spec Specifies the pseudorandom function (PRF) used to generate keying

material, the bulk data encryption algorithm (such as null, AES, etc.) and the MAC algorithm (such as HMAC-SHA1). It also defines cryptographic attributes such as the mac_length. (See Appendix A.6 for formal definition.)

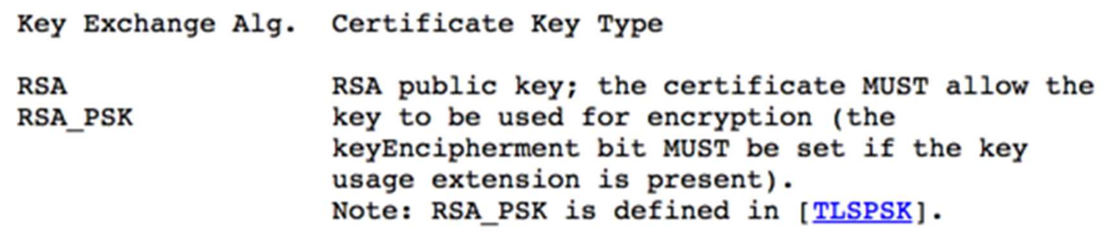
master secret
48-byte secret shared between the client and server.

is resumable
A flag indicating whether the session can be used to initiate new connections.

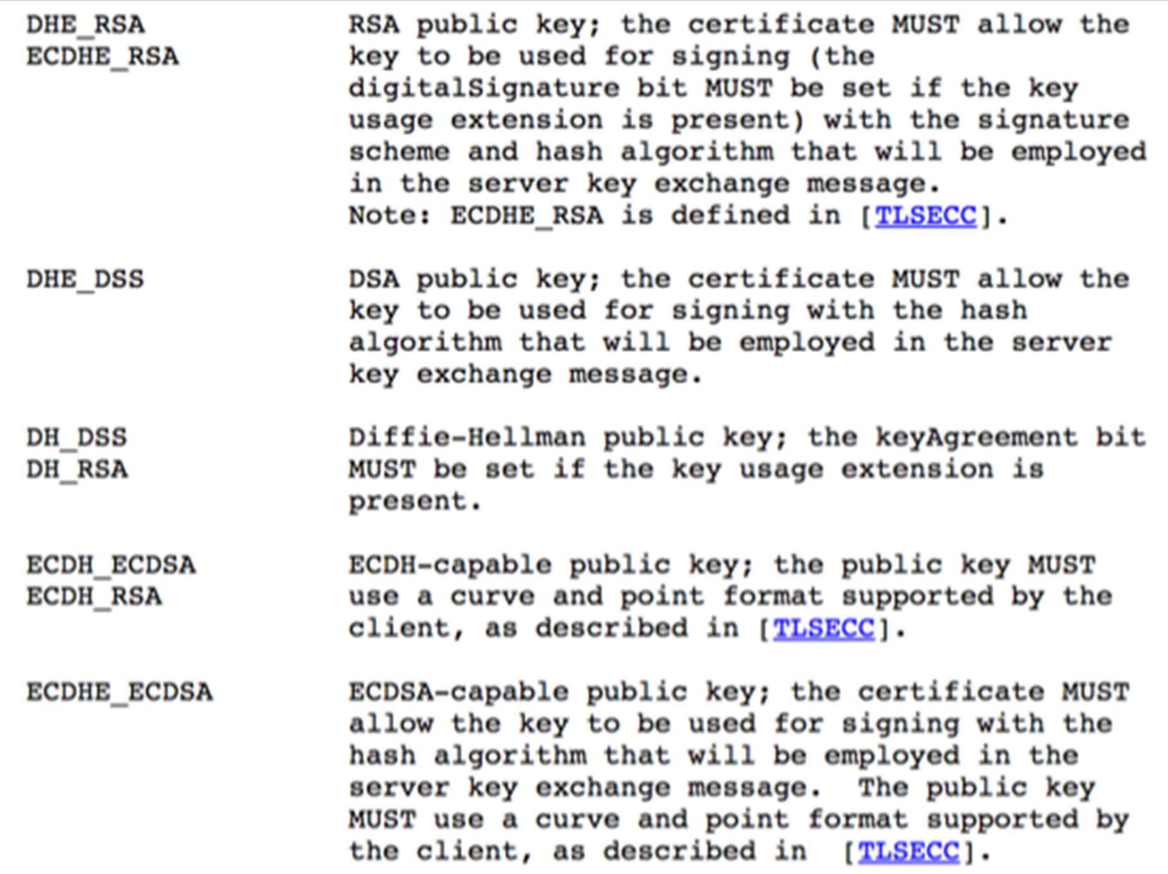
These items are then used to create security parameters for use by the record layer when protecting application data. Many connections can be instantiated using the same session through the resumption feature of the TLS Handshake Protocol.

Source: The Transport Layer Security (TLS) Protocol Version 1.2, IETF RFC 5246, https://tools.ietf.org/html/rfc5246, Last accessed on Mar 19, 2020

As explained above, Asana and Apple negotiate a key exchange algorithm from among the following key exchange algorithms:

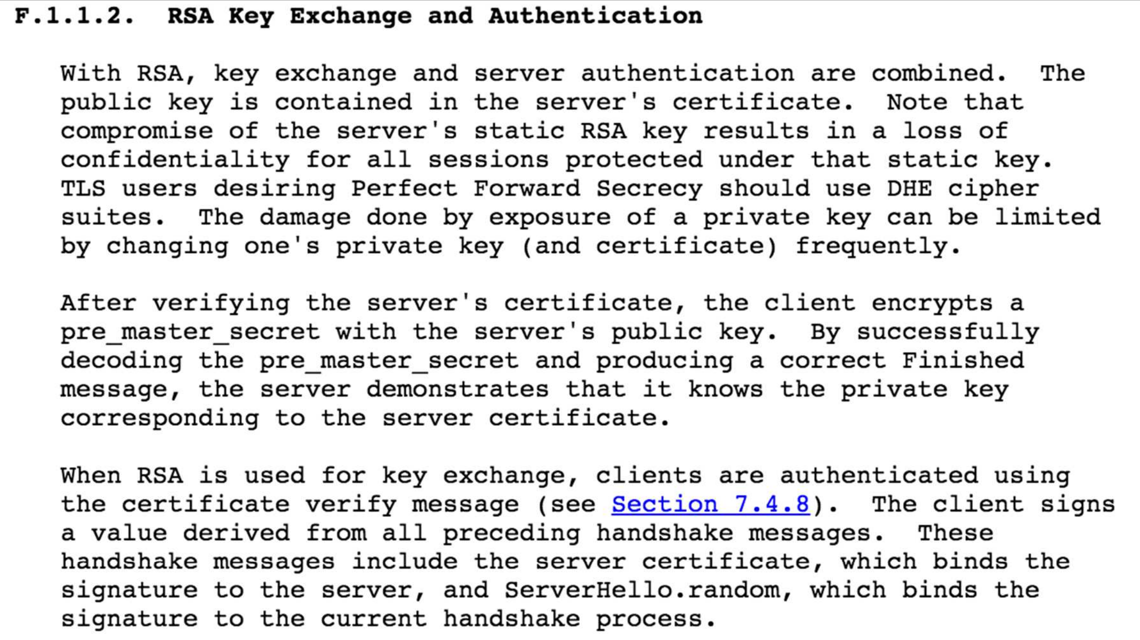


```
Key Exchange Alg.  Certificate Key Type

RSA                RSA public key; the certificate MUST allow the
RSA_PSK            key to be used for encryption (the
                   keyEncipherment bit MUST be set if the key
                   usage extension is present).
                   Note: RSA_PSK is defined in [TLSPSK].
```



```
DHE_RSA                 RSA public key; the certificate MUST allow the
ECDHE_RSA               key to be used for signing (the
                        digitalSignature bit MUST be set if the key
                        usage extension is present) with the signature
                        scheme and hash algorithm that will be employed
                        in the server key exchange message.
                        Note: ECDHE_RSA is defined in [TLSECC].

DHE_DSS                 DSA public key; the certificate MUST allow the
                        key to be used for signing with the hash
                        algorithm that will be employed in the server
                        key exchange message.

DH_DSS                  Diffie-Hellman public key; the keyAgreement bit
DH_RSA                  MUST be set if the key usage extension is
                        present.

ECDH_ECDSA              ECDH-capable public key; the public key MUST
ECDH_RSA                use a curve and point format supported by the
                        client, as described in [TLSECC].

ECDHE_ECDSA             ECDSA-capable public key; the certificate MUST
                        allow the key to be used for signing with the
                        hash algorithm that will be employed in the
                        server key exchange message.  The public key
                        MUST use a curve and point format supported by
                        the client, as described in  [TLSECC].
```

Source: https://tools.ietf.org/html/rfc5246 at 48-49, Last accessed on Mar 19, 2020

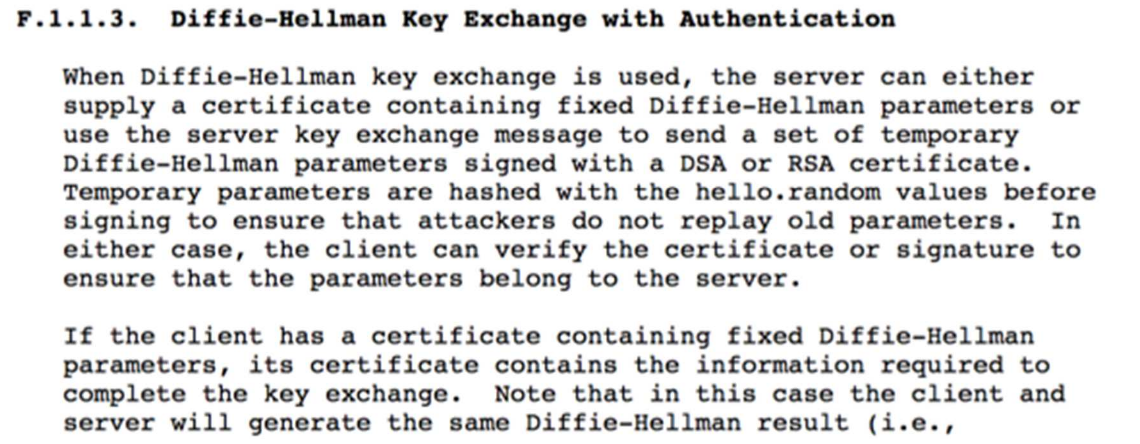


F.1.1.2. RSA Key Exchange and Authentication

With RSA, key exchange and server authentication are combined. The public key is contained in the server's certificate. Note that compromise of the server's static RSA key results in a loss of confidentiality for all sessions protected under that static key. TLS users desiring Perfect Forward Secrecy should use DHE cipher suites. The damage done by exposure of a private key can be limited by changing one's private key (and certificate) frequently.

After verifying the server's certificate, the client encrypts a pre_master_secret with the server's public key. By successfully decoding the pre_master_secret and producing a correct Finished message, the server demonstrates that it knows the private key corresponding to the server certificate.

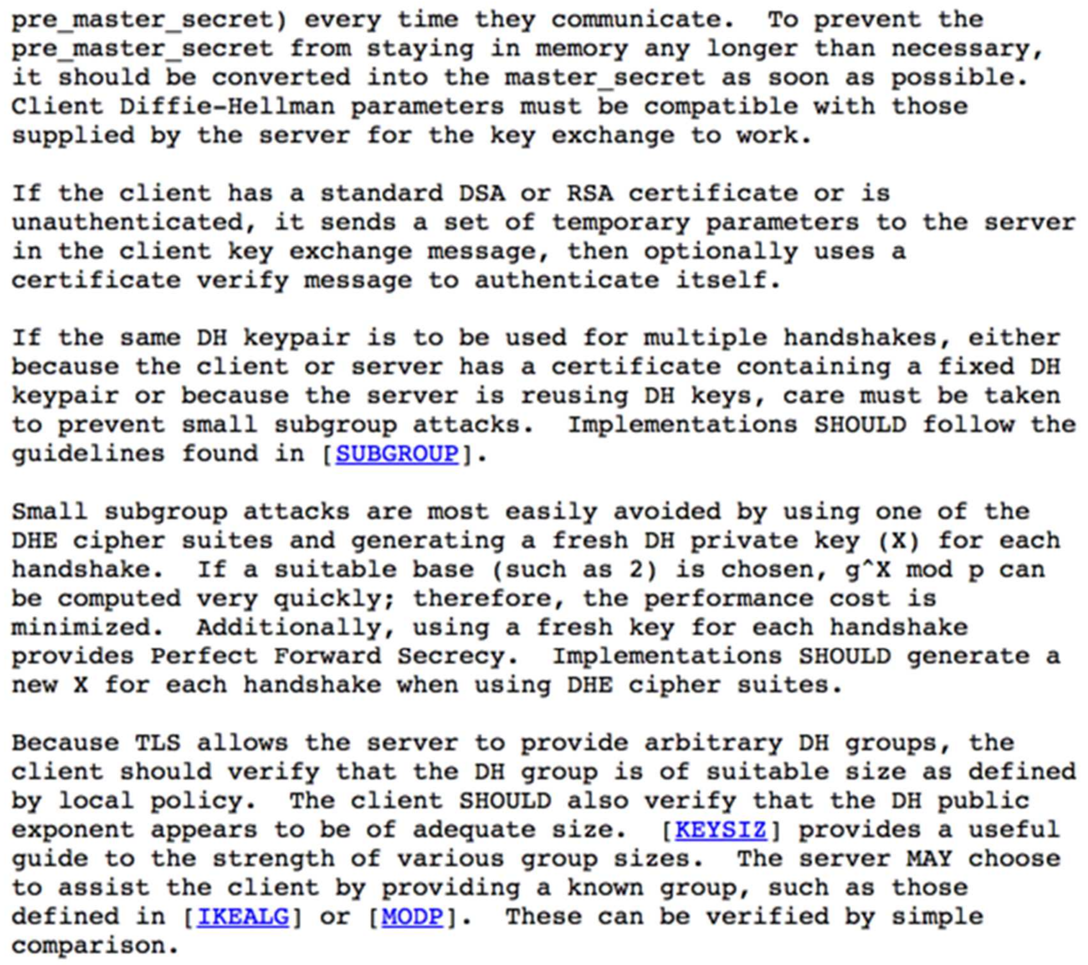
When RSA is used for key exchange, clients are authenticated using the certificate verify message (see Section 7.4.8). The client signs a value derived from all preceding handshake messages. These handshake messages include the server certificate, which binds the signature to the server, and ServerHello.random, which binds the signature to the current handshake process.



F.1.1.3. Diffie-Hellman Key Exchange with Authentication

When Diffie-Hellman key exchange is used, the server can either supply a certificate containing fixed Diffie-Hellman parameters or use the server key exchange message to send a set of temporary Diffie-Hellman parameters signed with a DSA or RSA certificate. Temporary parameters are hashed with the hello.random values before signing to ensure that attackers do not replay old parameters. In either case, the client can verify the certificate or signature to ensure that the parameters belong to the server.

If the client has a certificate containing fixed Diffie-Hellman parameters, its certificate contains the information required to complete the key exchange. Note that in this case the client and server will generate the same Diffie-Hellman result (i.e.,



```
pre_master_secret) every time they communicate.  To prevent the
pre_master_secret from staying in memory any longer than necessary,
it should be converted into the master_secret as soon as possible.
Client Diffie-Hellman parameters must be compatible with those
supplied by the server for the key exchange to work.

If the client has a standard DSA or RSA certificate or is
unauthenticated, it sends a set of temporary parameters to the server
in the client key exchange message, then optionally uses a
certificate verify message to authenticate itself.

If the same DH keypair is to be used for multiple handshakes, either
because the client or server has a certificate containing a fixed DH
keypair or because the server is reusing DH keys, care must be taken
to prevent small subgroup attacks.  Implementations SHOULD follow the
guidelines found in [SUBGROUP].

Small subgroup attacks are most easily avoided by using one of the
DHE cipher suites and generating a fresh DH private key (X) for each
handshake.  If a suitable base (such as 2) is chosen, g^X mod p can
be computed very quickly; therefore, the performance cost is
minimized.  Additionally, using a fresh key for each handshake
provides Perfect Forward Secrecy.  Implementations SHOULD generate a
new X for each handshake when using DHE cipher suites.

Because TLS allows the server to provide arbitrary DH groups, the
client should verify that the DH group is of suitable size as defined
by local policy.  The client SHOULD also verify that the DH public
exponent appears to be of adequate size.  [KEYSIZ] provides a useful
guide to the strength of various group sizes.  The server MAY choose
to assist the client by providing a known group, such as those
defined in [IKEALG] or [MODP].  These can be verified by simple
comparison.
```

Source: The Transport Layer Security (TLS) Protocol Version 1.2, IETF RFC 5246, https://tools.ietf.org/html/rfc5246, Last accessed on Mar 19, 2020

For each of the above key exchange algorithms, Apple and/or Asana encrypts predetermined data using the generated public key(s).

For **RSA and RSA _PSK**, Asana encrypts a random premaster secret with Apple's RSA public key and sends the encrypted premaster secret to Apple. Apple decrypts the premaster secret with its matched RSA private key. Asana and Apple both use the premaster secret to compute a master secret which is then used by both Asana and Apple to encrypt all subsequent communications between Asana and Apple. Therefore, when any of **RSA and**

	RSA _PSK algorithms are chosen during a TLS handshake, Asana encrypts predetermined data (i.e. the premaster secret) with the generated public key (i.e. Apple's RSA public key). For the other key exchange algorithms, namely Diffie-Hellman based algorithms such as **DHE_RSA, ECDHE_RSA, DH_RSA, DHE_DSS, DH_DSS, ECDH_RSA**, **ECDH_ECDSA and ECDHE_ECDSA,** Asana sends its Diffie-Hellman public key[9] to Apple while Apple sends its Diffie-Hellman public key to Asana. Asana then uses Apple's Diffie-Hellman public key combined with Asana's own Diffie-Hellman private key to compute a premaster secret. Apple in turn uses Asana's Diffie-Hellman public key combined with Apple's own Diffie-Hellman private key to compute the same premaster secret. Asana and Apple both use the premaster secret to compute a master secret[10] which is then used by both Asana and Apple to encrypt all subsequent communications between Asana and Apple. Therefore, when any of the Diffie-Hellman based key exchange algorithms are chosen during a TLS handshake, Asana encrypts predetermined data, i.e. all communication with Apple subsequent to the TLS handshake, including at least the executable compiled code related to its mobile apps and information such as name, category, screenshots and description related to Asana's mobile apps, with the generated public key, i.e. Apple's Diffie-Hellman public key, since the master secret used to encrypt the predetermined data is a combination of Apple's Diffie-Hellman public key and Asana's Diffie-Hellman private key[11]. Similarly, when any of the Diffie-Hellman based key exchange algorithms are chosen during a TLS handshake, Apple encrypts predetermined data, i.e. all communication with Apple subsequent to the TLS handshake, including at least textual and graphic data and software relating to Apple App Store Connect website and forms that Asana uses for uploading its mobile app to Apple App Store, with the generated public key, i.e. Asana's Diffie-Hellman public key, since the master secret used to encrypt the predetermined data is a combination of Asana's Diffie-Hellman public key and Apple's Diffie-Hellman private key[12].
and building an executable tamper resistant key module identified for a selected program, the executable	Relevant to this claim element is the Court's previous construction[13] of "executable tamper resistant key module" / "executable tamper resistant code module" / "tamper resistant key module" to mean "software that is designed to work with other software, that is resistant to observation and

[9] For Diffie-Hellman (DH) based algorithms such as DH_RSA, DHE_RSA, ECDH_RSA, ECDHE_RSA, DH_DSS, DHE_DSS, ECDH_ECDSA and ECDHE_ECDSA, Apple calculates a hash of the message containing their Diffie-Hellman public key and encrypts the hash with their RSA/DSA/ECDSA private key (i.e. signing the message). Apple then sends that RSA/DSA/ECDSA public key to Asana in a digital certificate so that Asana can authenticate the Apple server by decrypting the hash using Apple's public key and matching the decryption result to a hash of the received message as calculated by Asana itself. If the two values match, Asana knows that the message originated from Apple and not from a malicious third party.

[10] The master secret obtained from the premaster secret may be hashed according to a hashing algorithm also specified in the cipher suite in order to remove weak bits, as explained in https://tools.ietf.org/html/rfc5246, Sections 6.3, 7.4.9 and 8.1.

[11] See, for example, https://tools.ietf.org/html/rfc2631, Page 2, Last accessed on Mar 19, 2020

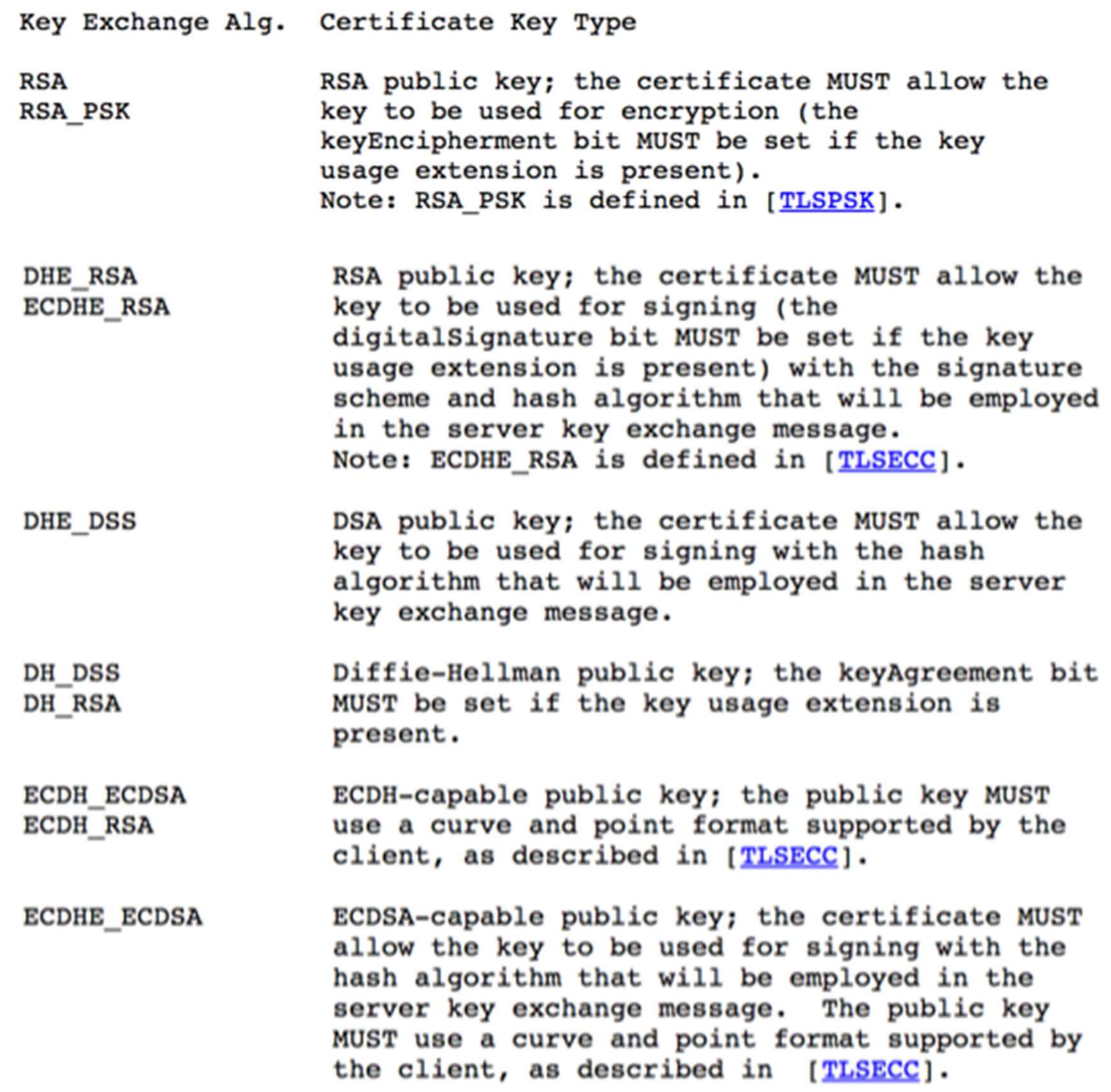
[12] Id.

[13] See Memorandum Opinion and Order, Document 104 signed by Judge Rodney Gilstrap on 7/21/2016 in re Plano Encryption Technologies, LLC v. American Bank of Texas (2:15-cv-01273).

tamper resistant key module including the generated private key and the encrypted predetermined data.	modification, and that includes a key for secure communication." Also relevant to this claim element is the Court's previous rejection of limiting "including" to compiling[14]. The method step of "building an executable tamper resistant key module identified for a selected program, the executable tamper resistant key module including the generated private key and the encrypted predetermined data" is performed by Asana and/or its agents. To the extent any portion of the method step is performed by Apple and/or its agents, such acts are attributable to Asana (i) because Asana works together with Apple in a joint enterprise in the building and distribution of its mobile apps, or (ii) because Apple distributes and markets Asana's mobile apps under the direction and control of Asana, or acts as agent, or on behalf of Asana, in the building, marketing and distribution of Asana's mobile apps. Building of an "executable tamper resistant code module" (that is, the mobile app) requires the inclusion of at least the following different asymmetric key pairs: (1) An asymmetric key pair must be included in order to send the mobile app from Asana to Apple securely by SSL/TLS; and (2) An asymmetric key pair must be included by Asana in order to digitally sign the mobile app with a private asymmetric key and to verify the mobile app has not been changed with the public key for iOS compatible mobile apps. The asymmetric key pair that is included to digitally sign the mobile app is different from and in addition to the claimed asymmetric key pair used to securely upload the mobile app to the Apple servers for distribution on the Apple App Store. Thus, at least one asymmetric key pair is included having the claimed public key and claimed private key in building the tamper resistant app so that the mobile app code can be sent uploaded to the Apple servers using SSL/TLS protocol. This asymmetric key pair(s) (as with the asymmetric key pair used to digitally sign the app) is complementary as described below by performing complementary functions, such as encrypting and decrypting data and/or creating and verifying signatures. As described in greater detail below, the claimed public and private key are generated and used to securely upload the mobile app onto the Apple servers by SSL/TLS when building the app, where the app includes the generated private key of the claimed asymmetric key pair(s) and the encrypted predetermined data. Asana uploads their mobile apps to Apple using a TLS connection – which begins with a TLS handshake. A TLS handshake is a mandatory procedure that allows Asana and Apple to exchange cryptographic parameters, including a cipher suite and arrive at a shared master secret for encrypting communication including upload of Asana's mobile apps to Apple servers.

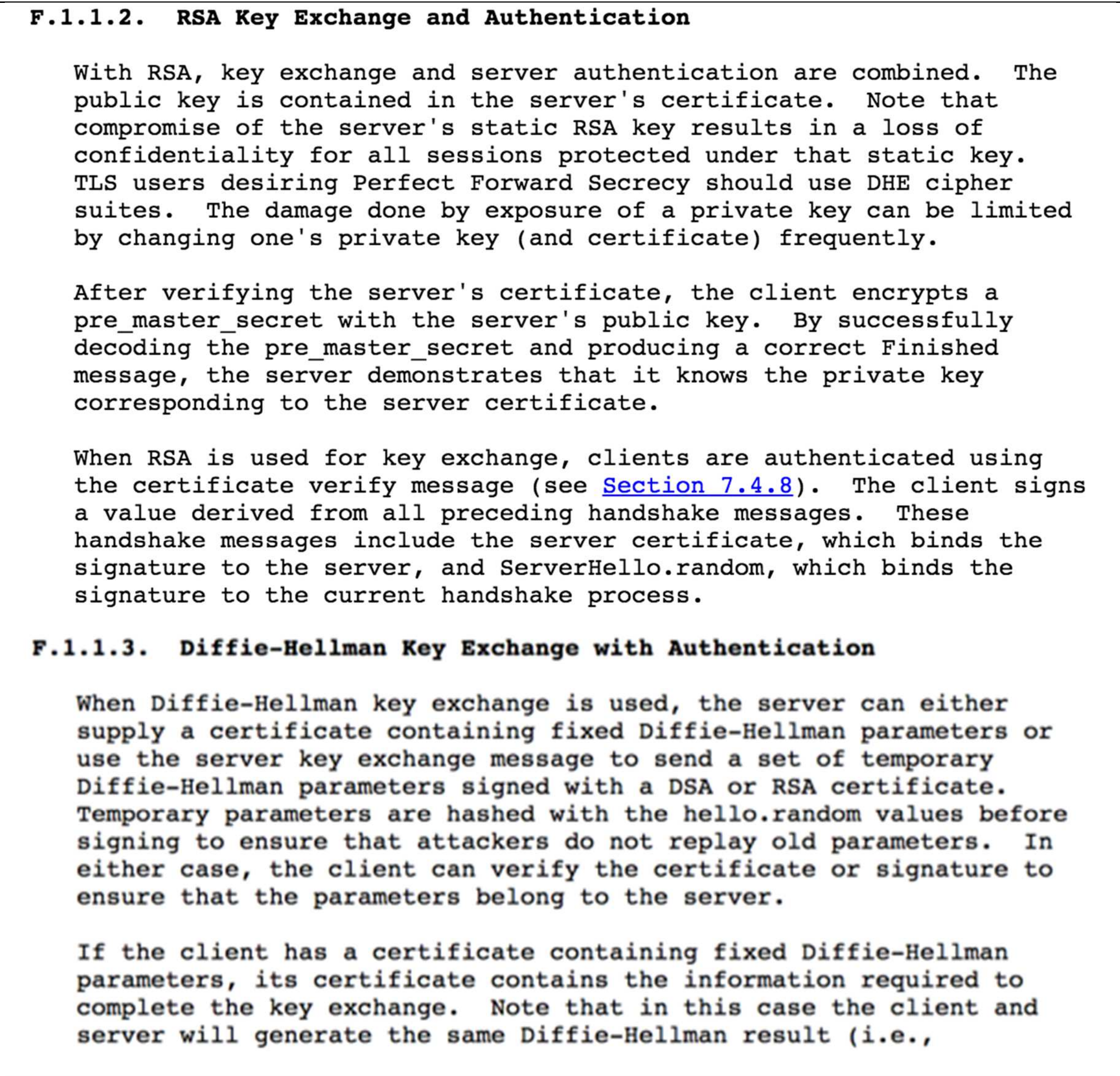
[14] Also relevant is the Court's construction of "an asymmetric key pair having a public key and a private key" in claims 1, 9 and 10 to mean "one or more asymmetric key pairs, one of the asymmetric key pairs having the claimed public key and claimed private key, the asymmetric keys of an asymmetric key pair being complementary by performing complementary functions, such as encrypting and decrypting data or creating and verifying signatures."

A TLS handshake begins with Asana sending a list of cipher suites supported by Asana to Apple. These cipher suites specify at least one or more of the following key exchange algorithms:



```
Key Exchange Alg.  Certificate Key Type

RSA                RSA public key; the certificate MUST allow the
RSA_PSK            key to be used for encryption (the
                   keyEncipherment bit MUST be set if the key
                   usage extension is present).
                   Note: RSA_PSK is defined in [TLSPSK].

DHE_RSA            RSA public key; the certificate MUST allow the
ECDHE_RSA          key to be used for signing (the
                   digitalSignature bit MUST be set if the key
                   usage extension is present) with the signature
                   scheme and hash algorithm that will be employed
                   in the server key exchange message.
                   Note: ECDHE_RSA is defined in [TLSECC].

DHE_DSS            DSA public key; the certificate MUST allow the
                   key to be used for signing with the hash
                   algorithm that will be employed in the server
                   key exchange message.

DH_DSS             Diffie-Hellman public key; the keyAgreement bit
DH_RSA             MUST be set if the key usage extension is
                   present.

ECDH_ECDSA         ECDH-capable public key; the public key MUST
ECDH_RSA           use a curve and point format supported by the
                   client, as described in [TLSECC].

ECDHE_ECDSA        ECDSA-capable public key; the certificate MUST
                   allow the key to be used for signing with the
                   hash algorithm that will be employed in the
                   server key exchange message.  The public key
                   MUST use a curve and point format supported by
                   the client, as described in  [TLSECC].
```

Source: https://tools.ietf.org/html/rfc5246 at 48-49, Last accessed on Mar 19, 2020



F.1.1.2. RSA Key Exchange and Authentication

With RSA, key exchange and server authentication are combined. The public key is contained in the server's certificate. Note that compromise of the server's static RSA key results in a loss of confidentiality for all sessions protected under that static key. TLS users desiring Perfect Forward Secrecy should use DHE cipher suites. The damage done by exposure of a private key can be limited by changing one's private key (and certificate) frequently.

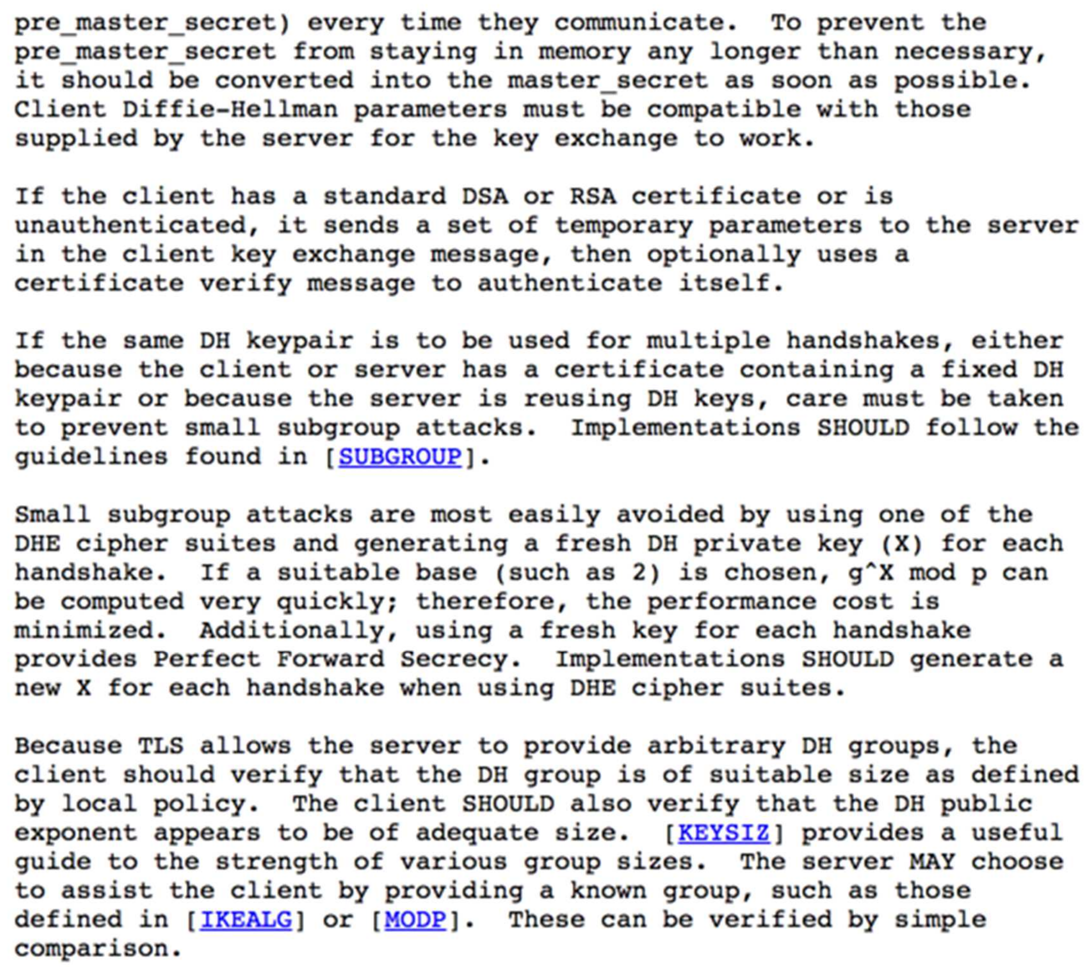
After verifying the server's certificate, the client encrypts a pre_master_secret with the server's public key. By successfully decoding the pre_master_secret and producing a correct Finished message, the server demonstrates that it knows the private key corresponding to the server certificate.

When RSA is used for key exchange, clients are authenticated using the certificate verify message (see Section 7.4.8). The client signs a value derived from all preceding handshake messages. These handshake messages include the server certificate, which binds the signature to the server, and ServerHello.random, which binds the signature to the current handshake process.

F.1.1.3. Diffie-Hellman Key Exchange with Authentication

When Diffie-Hellman key exchange is used, the server can either supply a certificate containing fixed Diffie-Hellman parameters or use the server key exchange message to send a set of temporary Diffie-Hellman parameters signed with a DSA or RSA certificate. Temporary parameters are hashed with the hello.random values before signing to ensure that attackers do not replay old parameters. In either case, the client can verify the certificate or signature to ensure that the parameters belong to the server.

If the client has a certificate containing fixed Diffie-Hellman parameters, its certificate contains the information required to complete the key exchange. Note that in this case the client and server will generate the same Diffie-Hellman result (i.e.,



```
pre_master_secret) every time they communicate.  To prevent the
pre_master_secret from staying in memory any longer than necessary,
it should be converted into the master_secret as soon as possible.
Client Diffie-Hellman parameters must be compatible with those
supplied by the server for the key exchange to work.

If the client has a standard DSA or RSA certificate or is
unauthenticated, it sends a set of temporary parameters to the server
in the client key exchange message, then optionally uses a
certificate verify message to authenticate itself.

If the same DH keypair is to be used for multiple handshakes, either
because the client or server has a certificate containing a fixed DH
keypair or because the server is reusing DH keys, care must be taken
to prevent small subgroup attacks.  Implementations SHOULD follow the
guidelines found in [SUBGROUP].

Small subgroup attacks are most easily avoided by using one of the
DHE cipher suites and generating a fresh DH private key (X) for each
handshake.  If a suitable base (such as 2) is chosen, g^X mod p can
be computed very quickly; therefore, the performance cost is
minimized.  Additionally, using a fresh key for each handshake
provides Perfect Forward Secrecy.  Implementations SHOULD generate a
new X for each handshake when using DHE cipher suites.

Because TLS allows the server to provide arbitrary DH groups, the
client should verify that the DH group is of suitable size as defined
by local policy.  The client SHOULD also verify that the DH public
exponent appears to be of adequate size.  [KEYSIZ] provides a useful
guide to the strength of various group sizes.  The server MAY choose
to assist the client by providing a known group, such as those
defined in [IKEALG] or [MODP].  These can be verified by simple
comparison.
```

Source: The Transport Layer Security (TLS) Protocol Version 1.2, IETF RFC 5246, https://tools.ietf.org/html/rfc5246, Last accessed on Mar 19, 2020

For each of the key exchange algorithms, Asana and/or Apple build an executable tamper resistant key module that includes the generated private key and the encrypted predetermined data.

	For **RSA and RSA _PSK**, the executable tamper resistant key module includes encrypted predetermined data (i.e. the encrypted premaster secret) as it would be impossible for Asana to send its mobile app to Apple using SSL/TLS without encrypting a premaster secret and sending it to Apple. The executable tamper resistant key module also includes: 1. Apple's RSA private key corresponding to Apple's RSA public key used by Asana to encrypt the premaster secret. 2. Asana's private key used to code-sign the mobile app. For **DHE_RSA, ECDHE_RSA, DH_RSA and ECDH_RSA**, the executable tamper resistant key module includes encrypted predetermined data (i.e. all communication with Apple subsequent to the TLS handshake, including at least the executable compiled code related to its mobile apps and information such as name, category, screenshots and description related to Asana's mobile apps). The executable tamper resistant key module also includes: 1. Apple's Diffie-Hellman private key corresponding to Apple's Diffie-Hellman public key used by Asana to compute the shared master secret. 2. Asana's Diffie-Hellman private key used to compute the shared master secret. 3. Apple's RSA private key used to sign the message containing Apple's Diffie-Hellman public key. 4. Asana's private key used to code-sign the mobile app. For **DHE_DSS and DH_DSS**, the executable tamper resistant key module includes encrypted predetermined data (i.e. all communication with Apple subsequent to the TLS handshake, including at least the executable compiled code related to its mobile apps and information such as name, category, screenshots and description related to Asana's mobile apps). The executable tamper resistant key module also includes: 1. Apple's Diffie-Hellman private key corresponding to Apple's Diffie-Hellman public key used by Asana to compute the shared master secret. 2. Asana's Diffie-Hellman private key used to compute the shared master secret. 3. Apple's DSA private key used to sign the message containing Apple's Diffie-Hellman public key. 4. Asana's private key used to code-sign the mobile app. For **ECDH_ECDSA and ECDHE_ECDSA**, the executable tamper resistant key module includes encrypted predetermined data (i.e. all communication with Apple subsequent to the TLS handshake, including at least the executable compiled code related to its mobile apps and information such as name, category, screenshots and description related to Asana's mobile apps). The executable tamper resistant key module also includes: 1. Apple's Diffie-Hellman private key corresponding to Apple's Diffie-Hellman public key used by Asana to compute the shared master secret. 2. Asana's Diffie-Hellman private key used to compute the shared master secret. 3. Apple's ECDSA private key used to sign the message containing Apple's Diffie-Hellman public key. 4. Asana's private key used to code-sign the mobile app.

Asana builds an executable tamper resistant key module identified for a selected program resident on a remote system. Specifically, Asana builds a mobile app, which is an executable tamper resistant key module, as explained in more detail below. This mobile app is identified for a selected program resident on a remote system, namely the iOS operating system on a remote mobile device.

The mobile app comprises an executable tamper resistant key module that is identified for either the iOS program and includes the claimed private key described above and the encrypted predetermined data encrypted with the claimed public key also described above in building the mobile app so that it can be made available for download from Apple servers onto devices compatible with iOS for use by customers of Asana. An asymmetric key pair is used not only to upload the binary code files for the mobile app, but an entire application package, including all of the metadata for the app, such as title, screenshots, and other resources or information such as application type, category, price, *etc*. which are included during the upload process so that the mobile app can be identified by potential users for download[15].

Asana's mobile app is each an executable tamper resistant key module because it is designed to work with other software, namely the iOS operating system as well as other applications or programs installed on a user's mobile device; because the mobile app is resistant to observation and modification, as explained below; and because in building Asana mobile apps on the Apple platform, Asana's apps include at least the claimed generated private key and the encrypted predetermined data including by way of example, the pre-master secret encrypted with the claimed generated public key when the mobile app is securely uploaded onto the Apple servers as described above.

The tamper resistant key module includes several keys "used for secure communications" per the Court's previous construction including at least the following:
For **RSA and RSA _PSK**:

1. Apple's RSA private key corresponding to Apple's RSA public key used by Asana to encrypt the premaster secret.
2. Asana's private key used to code-sign the mobile app.
3. Symmetric key used for uploading the mobile app to Apple subsequent to the TLS handshake.
4. Asymmetric keys and symmetric keys used for TLS communications during operation of the app.

For **DHE_RSA, ECDHE_RSA, DH_RSA and ECDH_RSA**:

1. Apple's Diffie-Hellman private key corresponding to Apple's Diffie-Hellman public key used by Asana to compute the shared master secret.
2. Asana's Diffie-Hellman private key used to compute the shared master secret.
3. Apple's RSA private key used to sign the message containing Apple's Diffie-Hellman public key.
4. Asana's private key used to code-sign the mobile app.
5. Shared master secret key used for uploading the mobile app to Apple subsequent to the TLS handshake.

[15] *See, e.g.,* https://developer.apple.com/library/ios/documentation/IDEs/Conceptual/AppDistributionGuide/SubmittingYourApp/SubmittingYourApp.html#//apple_ref/doc/uid/TP40012582-CH9-SW1, Last accessed on Mar 19, 2020

6. Asymmetric keys and symmetric keys used for TLS communications during operation of the app.

For **DHE_DSS and DH_DSS**:

1. Apple's Diffie-Hellman private key corresponding to Apple's Diffie-Hellman public key used by Asana to compute the shared master secret.
2. Asana's Diffie-Hellman private key used to compute the shared master secret.
3. Apple's DSA private key used to sign the message containing Apple's Diffie-Hellman public key.
4. Asana's private key used to code-sign the mobile app.
5. Shared master secret key used for uploading the mobile app to Apple subsequent to the TLS handshake.
6. Asymmetric keys and symmetric keys used for TLS communications during operation of the app.

For **ECDH_ECDSA and ECDHE_ECDSA**:

1. Apple's Diffie-Hellman private key corresponding to Apple's Diffie-Hellman public key used by Asana to compute the shared master secret.
2. Asana's Diffie-Hellman private key used to compute the shared master secret.
3. Apple's ECDSA private key used to sign the message containing Apple's Diffie-Hellman public key.
4. Asana's private key used to code-sign the mobile app.
5. Shared master secret key used for uploading the mobile app to Apple subsequent to the TLS handshake.
6. Asymmetric keys and symmetric keys used for TLS communications during operation of the app.

Asana's mobile apps are tamper resistant, resistant to observation and modification, as follows:

1. **Resistant to Observation Because App Source Code Is Compiled Before Upload**

Asana mobile apps are resistant to observation, at least in part, since Asana compiles its mobile app source code before submitting the app to Apple – and uploads the binary output of the compilation process rather than the source code itself[16].

See, e.g., App Store Connect Help**, "**Submit your app for review**"** stating **"**You submit your app to App Review to start the review process for making your app available on the App Store. However, before you can submit an app to App Review, you must provide the required metadata and choose the build for the version.
Before you submit an app to App Review, you choose which build (from all the builds you uploaded for a version) that you want to submit. You can

[16] *See, e.g.,* https://help.apple.com/app-store-connect/#/dev301cb2b3e, Last accessed on Mar 19, 2020

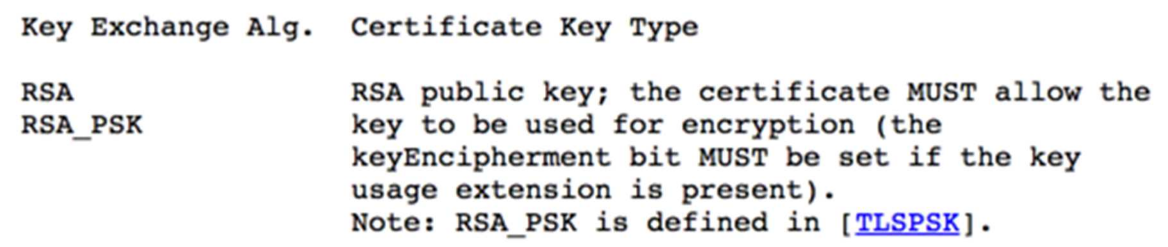
associate only one build with an App Store version. However, you can change the build as often as you want until you submit the version to App Review."

Source: https://help.apple.com/app-store-connect/#/dev301cb2b3e, Last accessed on Mar 19, 2020

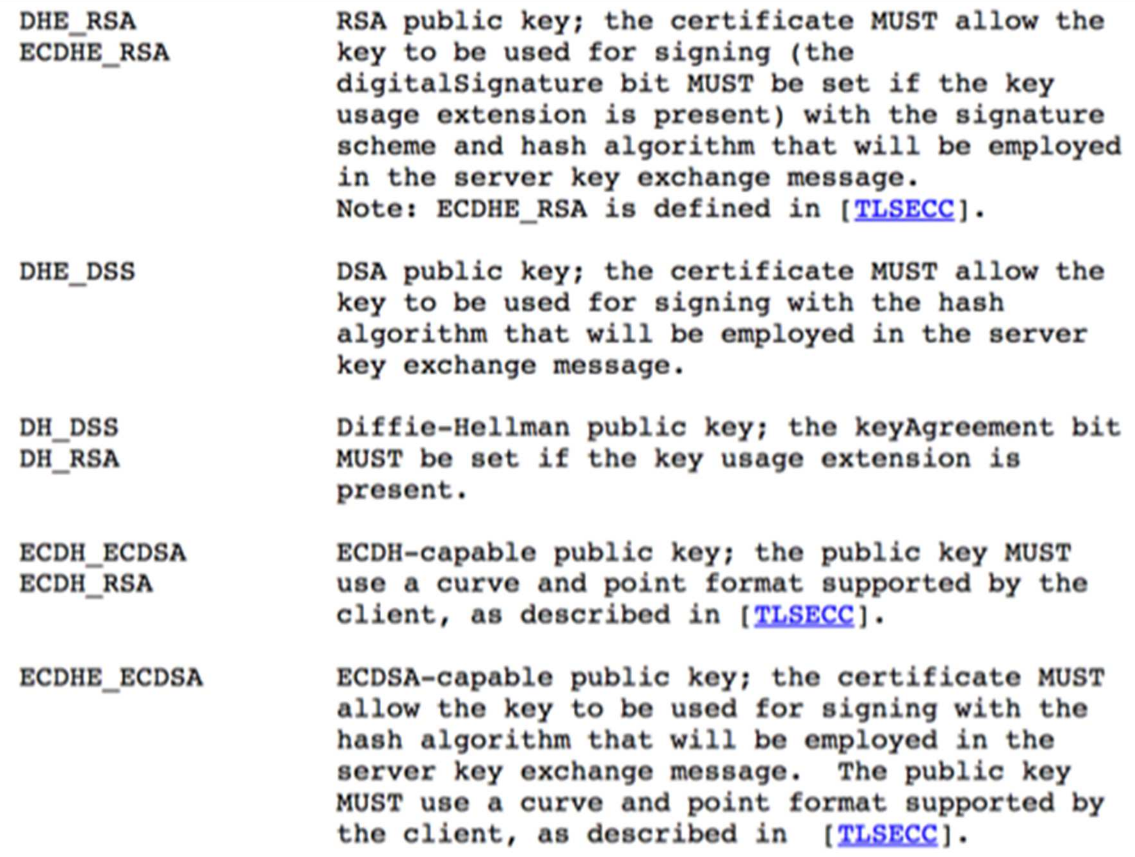
2. Resistant to Observation Because Upload To Apple Is Over SSL/TLS

Asana's mobile apps are made further resistant to observation, at least in part, because the mobile app is securely sent by SSL/TLS to Apple as part of the building process. App Store Connect portal (https://appstoreconnect.apple.com, Last accessed on Mar 19, 2020) establishes SSL/TLS communications when uploading Asana's apps, as evidenced by the "https" in its URL. Sending the mobile app code by SSL/TLS is necessary to keep the code from being observed in transit from the code developer to Apple.

The secure upload process starts with a TLS handshake procedure which uses asymmetric key encryption and necessitates generation of at least one asymmetric key pair. Specifically, Asana negotiates with Apple the cipher suite and the key exchange algorithm that will be used for the handshake.

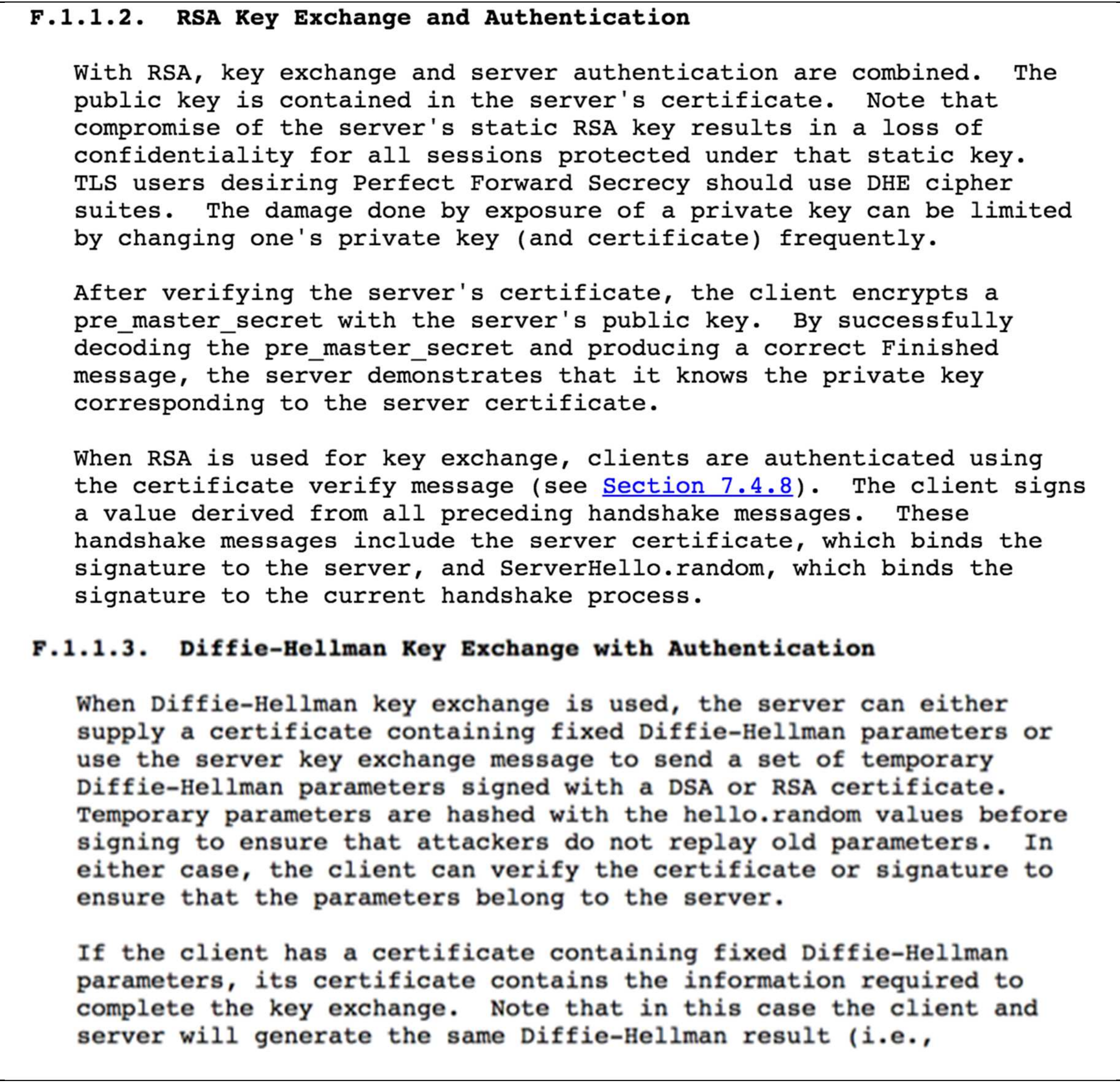


```
Key Exchange Alg.  Certificate Key Type

RSA                RSA public key; the certificate MUST allow the
RSA_PSK            key to be used for encryption (the
                   keyEncipherment bit MUST be set if the key
                   usage extension is present).
                   Note: RSA_PSK is defined in [TLSPSK].
```



```
DHE_RSA            RSA public key; the certificate MUST allow the
ECDHE_RSA          key to be used for signing (the
                   digitalSignature bit MUST be set if the key
                   usage extension is present) with the signature
                   scheme and hash algorithm that will be employed
                   in the server key exchange message.
                   Note: ECDHE_RSA is defined in [TLSECC].

DHE_DSS            DSA public key; the certificate MUST allow the
                   key to be used for signing with the hash
                   algorithm that will be employed in the server
                   key exchange message.

DH_DSS             Diffie-Hellman public key; the keyAgreement bit
DH_RSA             MUST be set if the key usage extension is
                   present.

ECDH_ECDSA         ECDH-capable public key; the public key MUST
ECDH_RSA           use a curve and point format supported by the
                   client, as described in [TLSECC].

ECDHE_ECDSA        ECDSA-capable public key; the certificate MUST
                   allow the key to be used for signing with the
                   hash algorithm that will be employed in the
                   server key exchange message.  The public key
                   MUST use a curve and point format supported by
                   the client, as described in  [TLSECC].
```

Source: https://tools.ietf.org/html/rfc5246 at 48-49, Last accessed on Mar 19, 2020



F.1.1.2. RSA Key Exchange and Authentication

With RSA, key exchange and server authentication are combined. The public key is contained in the server's certificate. Note that compromise of the server's static RSA key results in a loss of confidentiality for all sessions protected under that static key. TLS users desiring Perfect Forward Secrecy should use DHE cipher suites. The damage done by exposure of a private key can be limited by changing one's private key (and certificate) frequently.

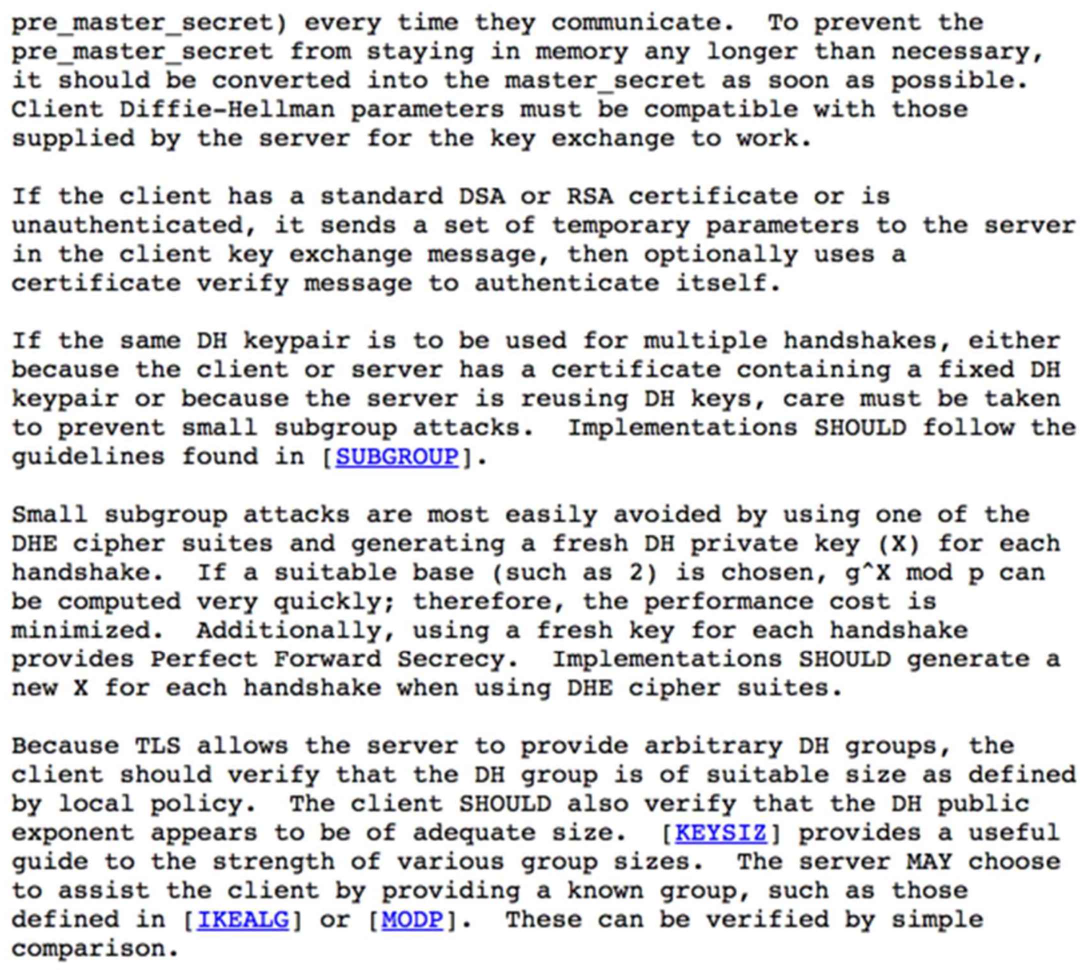
After verifying the server's certificate, the client encrypts a pre_master_secret with the server's public key. By successfully decoding the pre_master_secret and producing a correct Finished message, the server demonstrates that it knows the private key corresponding to the server certificate.

When RSA is used for key exchange, clients are authenticated using the certificate verify message (see Section 7.4.8). The client signs a value derived from all preceding handshake messages. These handshake messages include the server certificate, which binds the signature to the server, and ServerHello.random, which binds the signature to the current handshake process.

F.1.1.3. Diffie-Hellman Key Exchange with Authentication

When Diffie-Hellman key exchange is used, the server can either supply a certificate containing fixed Diffie-Hellman parameters or use the server key exchange message to send a set of temporary Diffie-Hellman parameters signed with a DSA or RSA certificate. Temporary parameters are hashed with the hello.random values before signing to ensure that attackers do not replay old parameters. In either case, the client can verify the certificate or signature to ensure that the parameters belong to the server.

If the client has a certificate containing fixed Diffie-Hellman parameters, its certificate contains the information required to complete the key exchange. Note that in this case the client and server will generate the same Diffie-Hellman result (i.e.,



```
pre_master_secret) every time they communicate.  To prevent the
pre_master_secret from staying in memory any longer than necessary,
it should be converted into the master_secret as soon as possible.
Client Diffie-Hellman parameters must be compatible with those
supplied by the server for the key exchange to work.

If the client has a standard DSA or RSA certificate or is
unauthenticated, it sends a set of temporary parameters to the server
in the client key exchange message, then optionally uses a
certificate verify message to authenticate itself.

If the same DH keypair is to be used for multiple handshakes, either
because the client or server has a certificate containing a fixed DH
keypair or because the server is reusing DH keys, care must be taken
to prevent small subgroup attacks.  Implementations SHOULD follow the
guidelines found in [SUBGROUP].

Small subgroup attacks are most easily avoided by using one of the
DHE cipher suites and generating a fresh DH private key (X) for each
handshake.  If a suitable base (such as 2) is chosen, g^X mod p can
be computed very quickly; therefore, the performance cost is
minimized.  Additionally, using a fresh key for each handshake
provides Perfect Forward Secrecy.  Implementations SHOULD generate a
new X for each handshake when using DHE cipher suites.

Because TLS allows the server to provide arbitrary DH groups, the
client should verify that the DH group is of suitable size as defined
by local policy.  The client SHOULD also verify that the DH public
exponent appears to be of adequate size.  [KEYSIZ] provides a useful
guide to the strength of various group sizes.  The server MAY choose
to assist the client by providing a known group, such as those
defined in [IKEALG] or [MODP].  These can be verified by simple
comparison.
```

Source: The Transport Layer Security (TLS) Protocol Version 1.2, IETF RFC 5246, https://tools.ietf.org/html/rfc5246, Last accessed on Mar 19, 2020

For each of the above key exchange algorithms, Apple and/or Asana encrypts all communication, including upload of the mobile app, using a master secret, rendering the communication resistant to observation during transit from Asana to Apple.

For **RSA and RSA _PSK**, Asana encrypts a random premaster secret with Apple's RSA public key and sends the encrypted premaster secret to Apple. Apple decrypts the premaster secret with its matched RSA private key. Asana and Apple both use the premaster secret to compute a master secret which is then used by both Asana and Apple to encrypt all subsequent communications between Asana and Apple.

For the other key exchange algorithms, namely Diffie-Hellman based algorithms such as **DHE_RSA, ECDHE_RSA, DH_RSA, DHE_DSS, DH_DSS, ECDH_RSA**, **ECDH_ECDSA and ECDHE_ECDSA,** Asana sends its Diffie-Hellman public key[17] to Apple while Apple sends its Diffie-Hellman public key to Asana. Asana then uses Apple's Diffie-Hellman public key combined with Asana's own Diffie-Hellman private key to compute a premaster secret. Apple in turn uses Asana's Diffie-Hellman public key combined with Apple's own Diffie-Hellman private key to compute the same premaster secret. Asana and Apple both use the premaster secret to compute a master secret which is then used by both Asana and Apple to encrypt all subsequent communications between Asana and Apple.

3. Resistant to Modification Because App Binary Is Code Signed

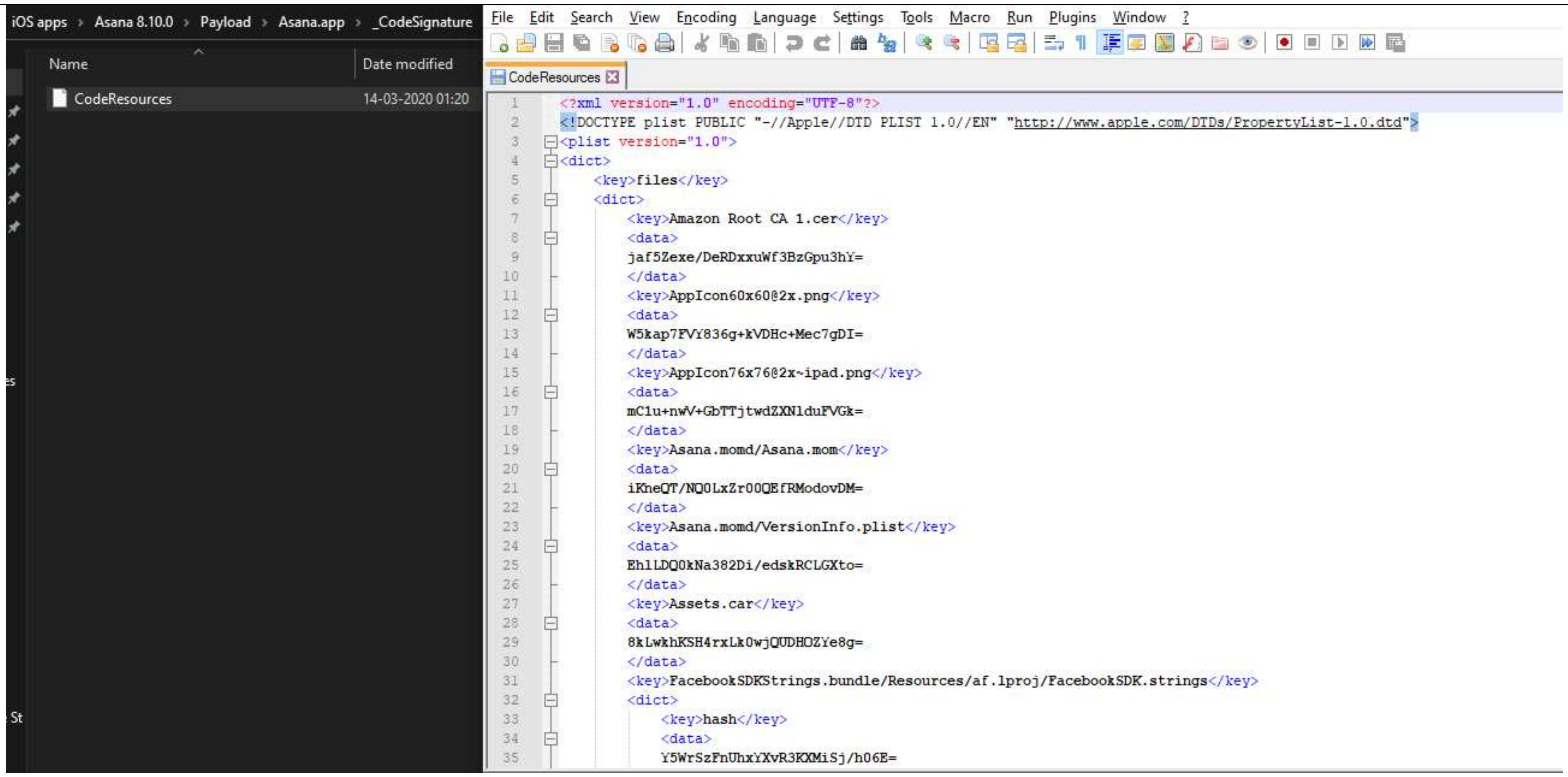
The mobile app code is made resistant to modification, at least in part, because the app binary is code signed. Apple dictates that each developer must sign the mobile app submission with his/her asymmetric developer key that certifies that the app has not been modified by a third party impersonator[18].

> *Xcode code signs your app during the build and archive process. If needed, Xcode requests a certificate and adds a signing certificate, the certificate with its public-private key pair, to your keychain. The certificate with the public key is added to your developer account.*
>
> Source: https://help.apple.com/xcode/mac/current/#/dev3a05256b8, Last accessed on Mar 19, 2020
>
> *A signing signing certificate includes the certificate with its public-private key pair issued by Apple, and is stored in your keychain. Because the private key is stored locally, protect it as you would an account password. An intermediate certificate is also required to be in your keychain to ensure that your certificate is issued by a certificate authority such as Apple.*
> *Your signing certificate is added to your keychain and the corresponding certificate is added to your developer account.*
>
> *Source:* https://help.apple.com/xcode/mac/current/#/dev1c7c2c67d, Last accessed on Mar 19, 2020

[17] For Diffie-Hellman (DH) based algorithms such as DH_RSA, DHE_RSA, ECDH_RSA, ECDHE_RSA, DH_DSS, DHE_DSS, ECDH_ECDSA and ECDHE_ECDSA, Apple calculates a hash of the message containing their Diffie-Hellman public key and encrypts the hash with their RSA/DSA/ECDSA private key (i.e. signing the message). Apple then sends that RSA/DSA/ECDSA public key to Asana in a digital certificate so that Asana can authenticate the Apple server by decrypting the hash using Apple's public key and matching the decryption result to a hash of the received message as calculated by Asana itself. If the two values match, Asana knows that the message originated from Apple and not from a malicious third party.

[18] *See, e.g.,* https://help.apple.com/xcode/mac/current/#/dev3a05256b8, Last accessed on Mar 19, 2020

	About Signing Identities and Certificates *Code signing (or signing) an app allows the system to identify who signed the app and to verify that the app has not been modified since it was signed.* *Signing is a requirement for uploading your app to App Store Connect and distributing it through TestFlight or the App Store. The operating system verifies the signature of apps downloaded from the App Store to ensure that apps with invalid signatures don't run. An app's executable code is protected by its signature because the signature becomes invalid if any of the executable code in the app bundle changes. A valid signature lets users trust that the app was signed by an Apple source and hasn't been modified since it was signed.* *Xcode uses your signing certificate to sign your app during the build process. The signing certificate consists of a public-private key pair and a certificate. The private key is used by cryptographic functions to generate the signature. The certificate is issued by Apple; it contains the public key and identifies you as the owner of the key pair. In order to sign apps, you must have both parts of your signing certificate, and an Apple certificate authority in your keychain.* *An app's signature can be removed, and the app can be re-signed using another signing certificate. For example, Apple re-signs all apps sold on the App Store. Also, a fully-tested development build of your app can be re-signed for submission to the App Store. Thus the signature is best understood not as proof of the app's origin but as a verifiable mark placed by the signer.* Source: https://help.apple.com/xcode/mac/current/#/devfbe995ebf, Last accessed on Mar 19, 2020 Asana complies with Apple's instructions on code signing as shown by Asana's mobile app contents. Asana's mobile apps contain files such as the file _CodeSignature/CodeResources in Asana's iOS apps which are generated during the code signing process as per instructions from Apple.


iOS apps › Asana 8.10.0 › Payload › Asana.app › _CodeSignature
Name
Date modified
CodeResources
14-03-2020 01:20
File Edit Search View Encoding Language Settings Tools Macro Run Plugins Window ?
CodeResources

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN" "http://www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
<dict>
    <key>files</key>
    <dict>
        <key>Amazon Root CA 1.cer</key>
        <data>
        jaf5Zexe/DeRDxxuWf3BzGpu3hY=
        </data>
        <key>AppIcon60x60@2x.png</key>
        <data>
        W5kap7FVY836g+kVDHc+Mec7gDI=
        </data>
        <key>AppIcon76x76@2x~ipad.png</key>
        <data>
        mC1u+nwV+GbTTjtwdZXNlduFVGk=
        </data>
        <key>Asana.momd/Asana.mom</key>
        <data>
        iKneQT/NQ0LxZr00QEfRModovDM=
        </data>
        <key>Asana.momd/VersionInfo.plist</key>
        <data>
        EhlLDQ0kNa382Di/edskRCLGXto=
        </data>
        <key>Assets.car</key>
        <data>
        8kLwkhKSH4rxLk0wjQUDHOZYe8g=
        </data>
        <key>FacebookSDKStrings.bundle/Resources/af.lproj/FacebookSDK.strings</key>
        <dict>
            <key>hash</key>
            <data>
            Y5WrSzFnUhxYXvR3KXMiSj/h06E=
```

Source: Contents of Asana: organize tasks & work, https://apps.apple.com/us/app/asana-organize-tasks-work/id489969512, as an example of Asana app, Last accessed on Mar 19, 2020

Asymmetrical key cryptography and hashing algorithms are used to create the unique digital signature for iOS mobile apps. The digital signature is used to sign the resources in an application package, including the compiled code. The private key of an asymmetric key pair that is generated for the digital code signing is used to code sign the app. This private key is included in the mobile app although the private key is not the claimed private key of the claimed generated asymmetric key pair because it does not match the claimed public key used to encrypt predetermined data, based on the court's construction.

	Hashes are created for every resource in the application package with the help of a hash algorithm. The signature manifest also has its own hash to prevent unauthorized changes. The hashes are encrypted with a private key. After the encryption is complete, the digital signature for the app is created. By signing the app binary with a digital signature, Asana's mobile apps are tamper resistant enabling Apple and the iOS mobile devices to verify that the application is being distributed by trusted source (*i.e.* Asana) and that the application has not been modified by a third party, which can be verified by the corresponding public key generated as part of the pair. Thus the app binary is made resistant to modification by digital signing. Accordingly Asana's iOS mobile apps establish SSL/TLS communications with Asana's servers, which involve a SSL/TLS handshake procedure involving asymmetric key encryption. SSL/TLS ensures secure communication and renders the mobile app data further resistant to observation.
2. The method of claim 1, wherein the program is on a remote system and further comprising sending the executable tamper resistant key module to the remote system.	Asana's mobile software application products and services including by way of example, but not limited to the following apps ("mobile applications", "mobile apps" or "Accused Products") that are specifically developed, used, sold, offered for sale, marketed, licensed and distributed by Asana to be downloaded onto Apple iOS mobile or tablet devices. • Asana: organize tasks & work (https://apps.apple.com/us/app/asana-organize-tasks-work/id489969512), Last accessed on Mar 19, 2020 Asana directly infringes and/or continues to knowingly induce Apple to infringe this claim by intentionally developing, making, marketing, advertising, providing, sending, distributing and licensing its mobile applications software, documentation, materials, training or support and aiding, abetting, encouraging, promoting or inviting use thereof. Upon information and belief, the method step of sending the executable tamper resistant key module to the remote system is performed by Apple and/or its agents – whose acts are attributable to Asana (i) because Asana works together with Apple in a joint enterprise in the building and distribution of its mobile apps, or (ii) because Apple distributes and markets Asana's mobile apps under the direction and control of Asana, or acts as agent, or on behalf of Asana, in the building, marketing and distribution of Asana's mobile apps. Alternatively, to the extent any portion of this method step is performed by Asana, such acts are attributable to Apple, who conditions participation in and the receipt of a benefit, namely, the distribution of Asana's mobile apps through its app store, upon compliance with certain mandatory procedures and guidelines dictated by Apple in the building and upload of Asana's mobile apps, and Asana induces infringement by Apple in the building, marketing and distribution of Asana's mobile apps. Asana's mobile apps are sent or downloaded from Apple servers and are executed on iOS remote devices such as mobile phones and tablets. When a user accesses Apple App Store – and requests to download Asana app, Apple sends the executable tamper resistant key module to the remote device(s).

Further, the step of "sending" Asana mobile apps to the remote system occurs via TLS/SSL communications. Thus, the sending of Asana mobile apps, to the extent required by the claims, also includes a private key and predetermined data encrypted by a public key, as explained in detail above.

In particular, Asana mobile apps sent to users' remote devices are tamper resistant, resistant to observation and modification as follows:

1. **Resistant to Observation Because App is Downloaded in Compiled Form**

Asana mobile apps are resistant to observation, at least in part, since Asana compiles its mobile app source code before submitting the app to Apple – and uploads the binary output of the compilation process rather than the source code itself – and hence a user can only download the compiled source code from Apple rather than the source code itself[19].

See, e.g., App Store Connect Help**, "**Submit your app for review" stating **"**You submit your app to App Review to start the review process for making your app available on the App Store. However, before you can submit an app to App Review, you must provide the required metadata and choose the build for the version.
Before you submit an app to App Review, you choose which build (from all the builds you uploaded for a version) that you want to submit. You can associate only one build with an App Store version. However, you can change the build as often as you want until you submit the version to App Review."

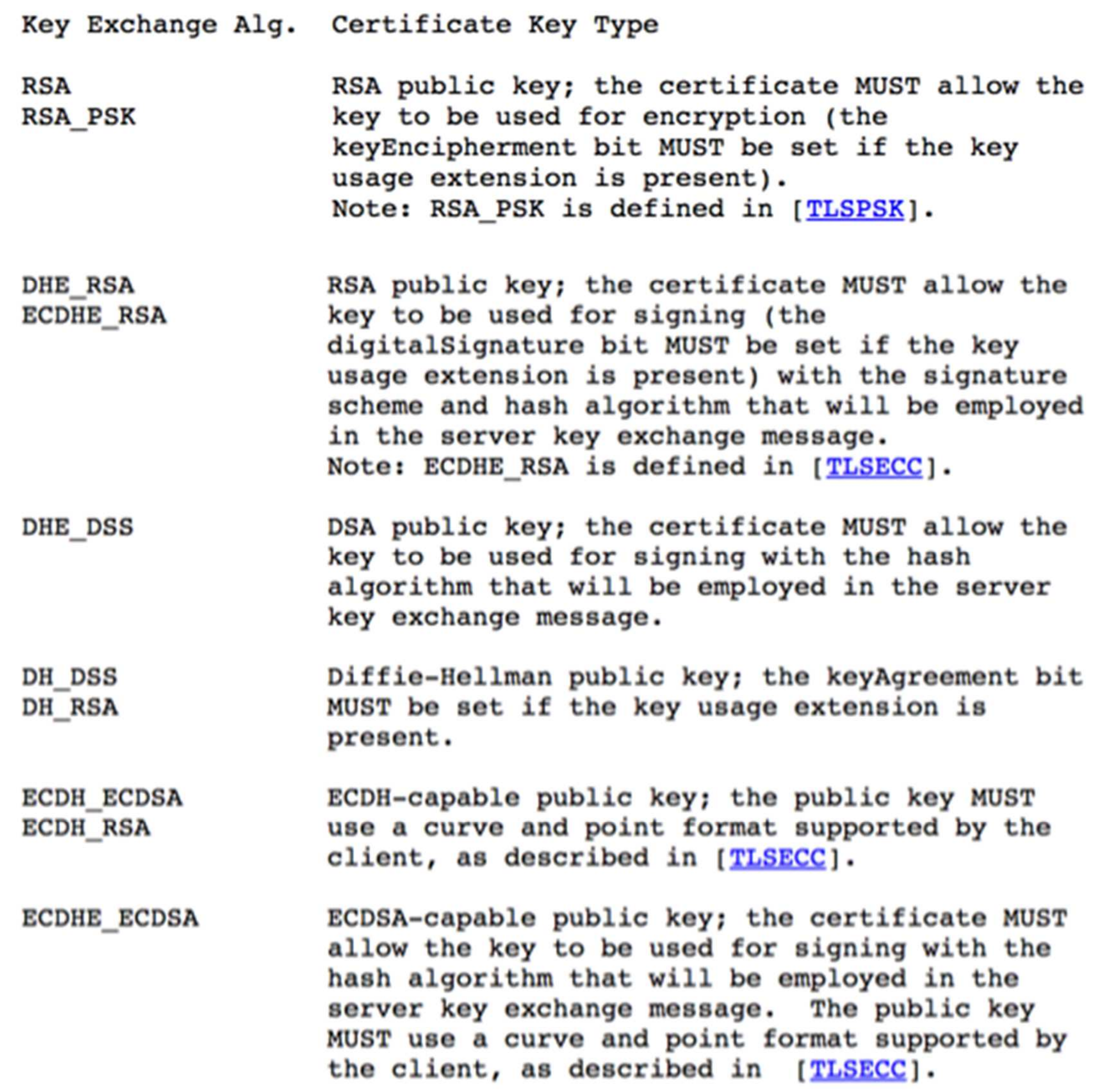
Source: https://help.apple.com/app-store-connect/#/dev301cb2b3e, Last accessed on Mar 19, 2020

2. **Resistant to Observation Because Download from Apple Is Over SSL/TLS**

Asana's mobile apps are made further resistant to observation, at least in part, because the mobile app is securely sent or downloaded by SSL/TLS from Apple servers. Asana app users establish SSL/TLS communications with Asana app store listings (for example, using the URL https://apps.apple.com/us/app/asana-organize-tasks-work/id489969512 for Asana: organize tasks & work, Last accessed on Mar 19, 2020) when downloading Asana's iOS apps, as evidenced by the "https" in the URL. Sending the mobile app code by SSL/TLS is necessary to keep the code from being observed in transit from Apple to the user's remote system.

The secure download process starts with a TLS handshake procedure which uses asymmetric key encryption and necessitates generation of at least one asymmetric key pair. Specifically, user's remote device negotiates with Apple the cipher suite and the key exchange algorithm that will be used for the handshake.

[19] *See, e.g.,* https://help.apple.com/app-store-connect/#/dev301cb2b3e, Last accessed on Mar 19, 2020

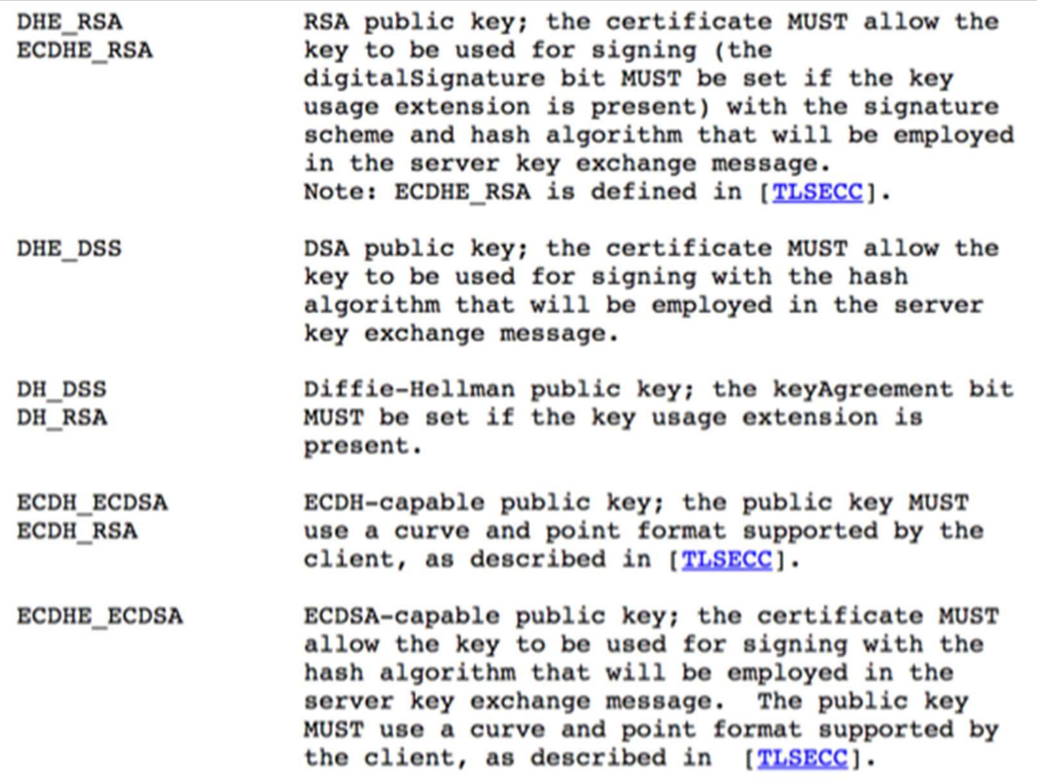


```
Key Exchange Alg.  Certificate Key Type

RSA                RSA public key; the certificate MUST allow the
RSA_PSK            key to be used for encryption (the
                   keyEncipherment bit MUST be set if the key
                   usage extension is present).
                   Note: RSA_PSK is defined in [TLSPSK].

DHE_RSA            RSA public key; the certificate MUST allow the
ECDHE_RSA          key to be used for signing (the
                   digitalSignature bit MUST be set if the key
                   usage extension is present) with the signature
                   scheme and hash algorithm that will be employed
                   in the server key exchange message.
                   Note: ECDHE_RSA is defined in [TLSECC].

DHE_DSS            DSA public key; the certificate MUST allow the
                   key to be used for signing with the hash
                   algorithm that will be employed in the server
                   key exchange message.

DH_DSS             Diffie-Hellman public key; the keyAgreement bit
DH_RSA             MUST be set if the key usage extension is
                   present.

ECDH_ECDSA         ECDH-capable public key; the public key MUST
ECDH_RSA           use a curve and point format supported by the
                   client, as described in [TLSECC].

ECDHE_ECDSA        ECDSA-capable public key; the certificate MUST
                   allow the key to be used for signing with the
                   hash algorithm that will be employed in the
                   server key exchange message.  The public key
                   MUST use a curve and point format supported by
                   the client, as described in  [TLSECC].
```

Source: https://tools.ietf.org/html/rfc5246 at 48-49, Last accessed on Mar 19, 2020

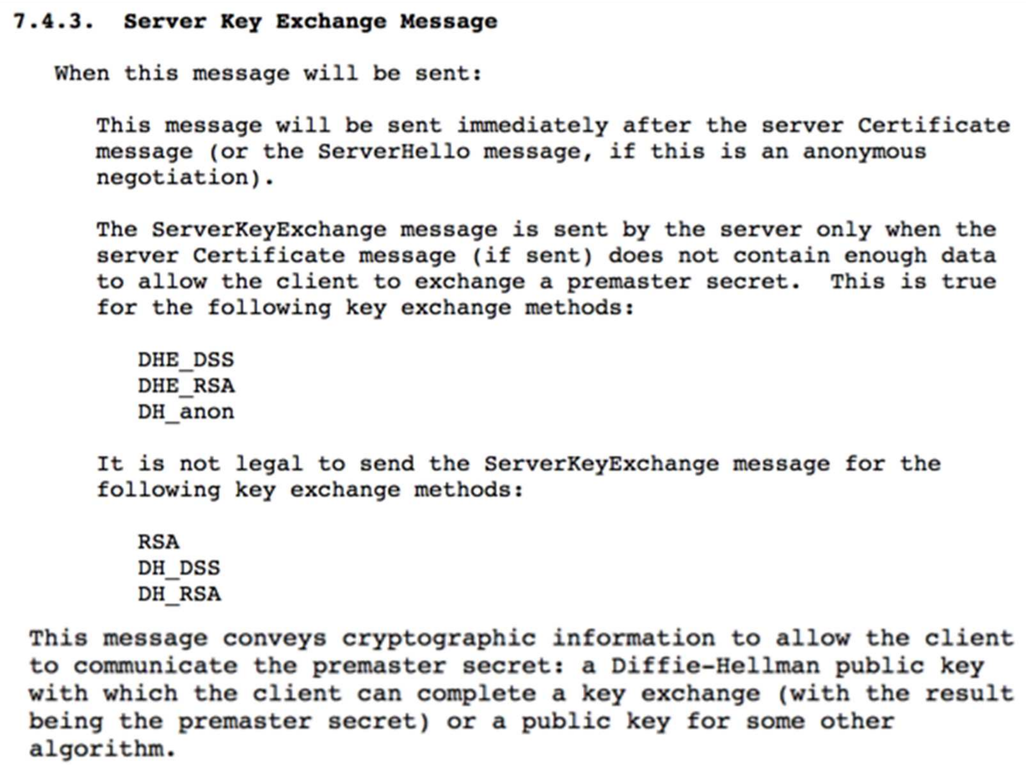
	Each of these algorithms necessitates generating one or more asymmetric key pairs – that are in turn used to compute a shared master secret for encrypting the mobile app download. For **RSA and RSA _PSK**, Apple server generates an RSA public-private key pair. For **DHE_RSA, ECDHE_RSA, DH_RSA and ECDH_RSA**, Apple server generates an RSA public-private key pair as well as a Diffie-Hellman public-private key pair[20]. The user's remote device also generates a second Diffie-Hellman public-private key pair. For **DHE_DSS and DH_DSS**, Apple server generates a DSA public-private key pair as well as a Diffie-Hellman public-private key pair. The user's remote device also generates a second Diffie-Hellman public-private key pair. For **ECDH_ECDSA and ECDHE_ECDSA**, Apple server generates an ECDSA public-private key pair as well as a Diffie-Hellman public-private key pair. The user's remote device also generates a second Diffie-Hellman public-private key pair.

[20] ECDH and ECDHE algorithms require generating at the server and the client, elliptical curve parameters that constitute a Diffie-Hellman public-private key pair. See, for example, https://tools.ietf.org/html/rfc5246 page 49-52, https://tools.ietf.org/html/rfc7525 page 12, http://www.cse.hut.fi/fi/opinnot/T-110.5241/2011/luennot-files/Network%20Security%2004%20-%20TLS.pdf, http://csrc.nist.gov/groups/STM/cmvp/documents/140-1/140sp/140sp2897.pdf, http://www.networkworld.com/article/2268575/lan-wan/chapter-2--ssl-vpn-technology.html, http://homes.esat.kuleuven.be/~fvercaut/papers/ACM2012.pdf, Implementing SSL / TLS Using Cryptography and PKI by Joshua Davies, ISBN 1118038770, 9781118038772, Page 305 and Introduction to Computer Networks and Cybersecurity By Chwan-Hwa (John) Wu, J. David Irwin, ISBN 1466572140, 9781466572140, Page 1021, which state that ECDH requires generating a Diffie-Hellman public-private key pair, Last accessed on Mar 19, 2020



```
DHE_RSA            RSA public key; the certificate MUST allow the
ECDHE_RSA          key to be used for signing (the
                   digitalSignature bit MUST be set if the key
                   usage extension is present) with the signature
                   scheme and hash algorithm that will be employed
                   in the server key exchange message.
                   Note: ECDHE_RSA is defined in [TLSECC].

DHE_DSS            DSA public key; the certificate MUST allow the
                   key to be used for signing with the hash
                   algorithm that will be employed in the server
                   key exchange message.

DH_DSS             Diffie-Hellman public key; the keyAgreement bit
DH_RSA             MUST be set if the key usage extension is
                   present.

ECDH_ECDSA         ECDH-capable public key; the public key MUST
ECDH_RSA           use a curve and point format supported by the
                   client, as described in [TLSECC].

ECDHE_ECDSA        ECDSA-capable public key; the certificate MUST
                   allow the key to be used for signing with the
                   hash algorithm that will be employed in the
                   server key exchange message.  The public key
                   MUST use a curve and point format supported by
                   the client, as described in  [TLSECC].
```

Source: https://tools.ietf.org/html/rfc5246 at 48-49, Last accessed on Mar 19, 2020



7.4.3. Server Key Exchange Message

When this message will be sent:

This message will be sent immediately after the server Certificate message (or the ServerHello message, if this is an anonymous negotiation).

The ServerKeyExchange message is sent by the server only when the server Certificate message (if sent) does not contain enough data to allow the client to exchange a premaster secret. This is true for the following key exchange methods:

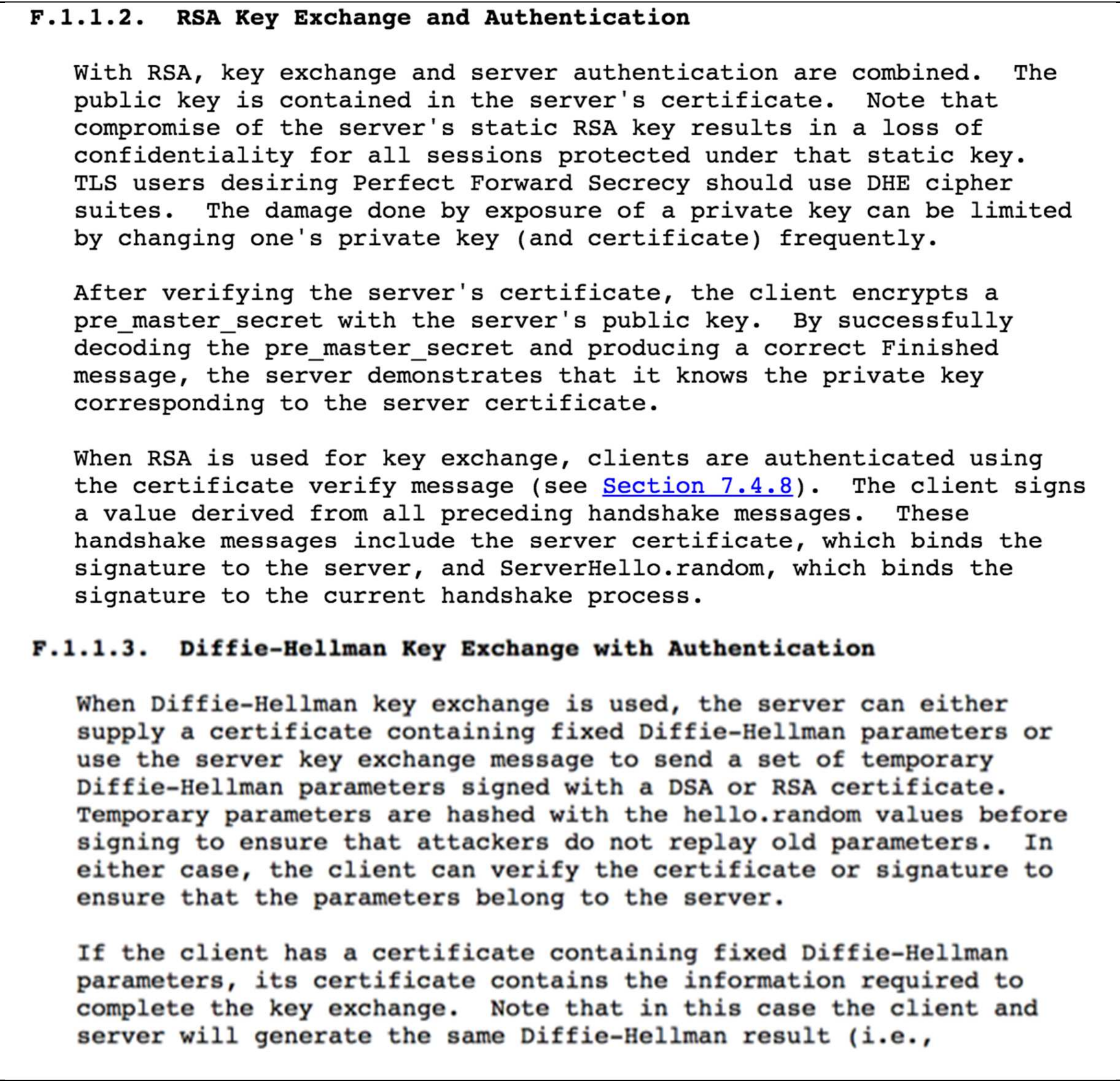
DHE_DSS
DHE_RSA
DH_anon

It is not legal to send the ServerKeyExchange message for the following key exchange methods:

RSA
DH_DSS
DH_RSA

This message conveys cryptographic information to allow the client to communicate the premaster secret: a Diffie-Hellman public key with which the client can complete a key exchange (with the result being the premaster secret) or a public key for some other algorithm.

Source: https://tools.ietf.org/html/rfc5246 at 50-51, Last accessed on Mar 19, 2020



F.1.1.2. RSA Key Exchange and Authentication

With RSA, key exchange and server authentication are combined. The public key is contained in the server's certificate. Note that compromise of the server's static RSA key results in a loss of confidentiality for all sessions protected under that static key. TLS users desiring Perfect Forward Secrecy should use DHE cipher suites. The damage done by exposure of a private key can be limited by changing one's private key (and certificate) frequently.

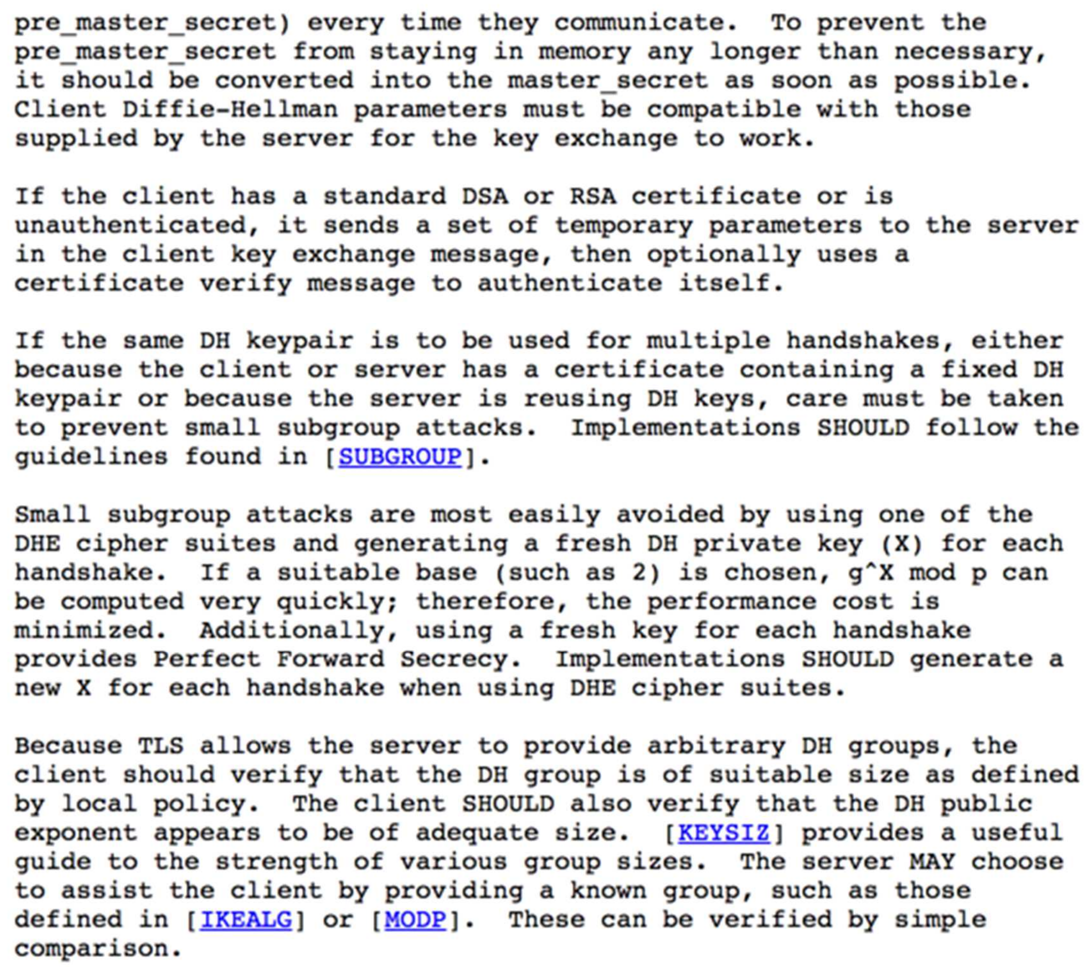
After verifying the server's certificate, the client encrypts a pre_master_secret with the server's public key. By successfully decoding the pre_master_secret and producing a correct Finished message, the server demonstrates that it knows the private key corresponding to the server certificate.

When RSA is used for key exchange, clients are authenticated using the certificate verify message (see Section 7.4.8). The client signs a value derived from all preceding handshake messages. These handshake messages include the server certificate, which binds the signature to the server, and ServerHello.random, which binds the signature to the current handshake process.

F.1.1.3. Diffie-Hellman Key Exchange with Authentication

When Diffie-Hellman key exchange is used, the server can either supply a certificate containing fixed Diffie-Hellman parameters or use the server key exchange message to send a set of temporary Diffie-Hellman parameters signed with a DSA or RSA certificate. Temporary parameters are hashed with the hello.random values before signing to ensure that attackers do not replay old parameters. In either case, the client can verify the certificate or signature to ensure that the parameters belong to the server.

If the client has a certificate containing fixed Diffie-Hellman parameters, its certificate contains the information required to complete the key exchange. Note that in this case the client and server will generate the same Diffie-Hellman result (i.e.,

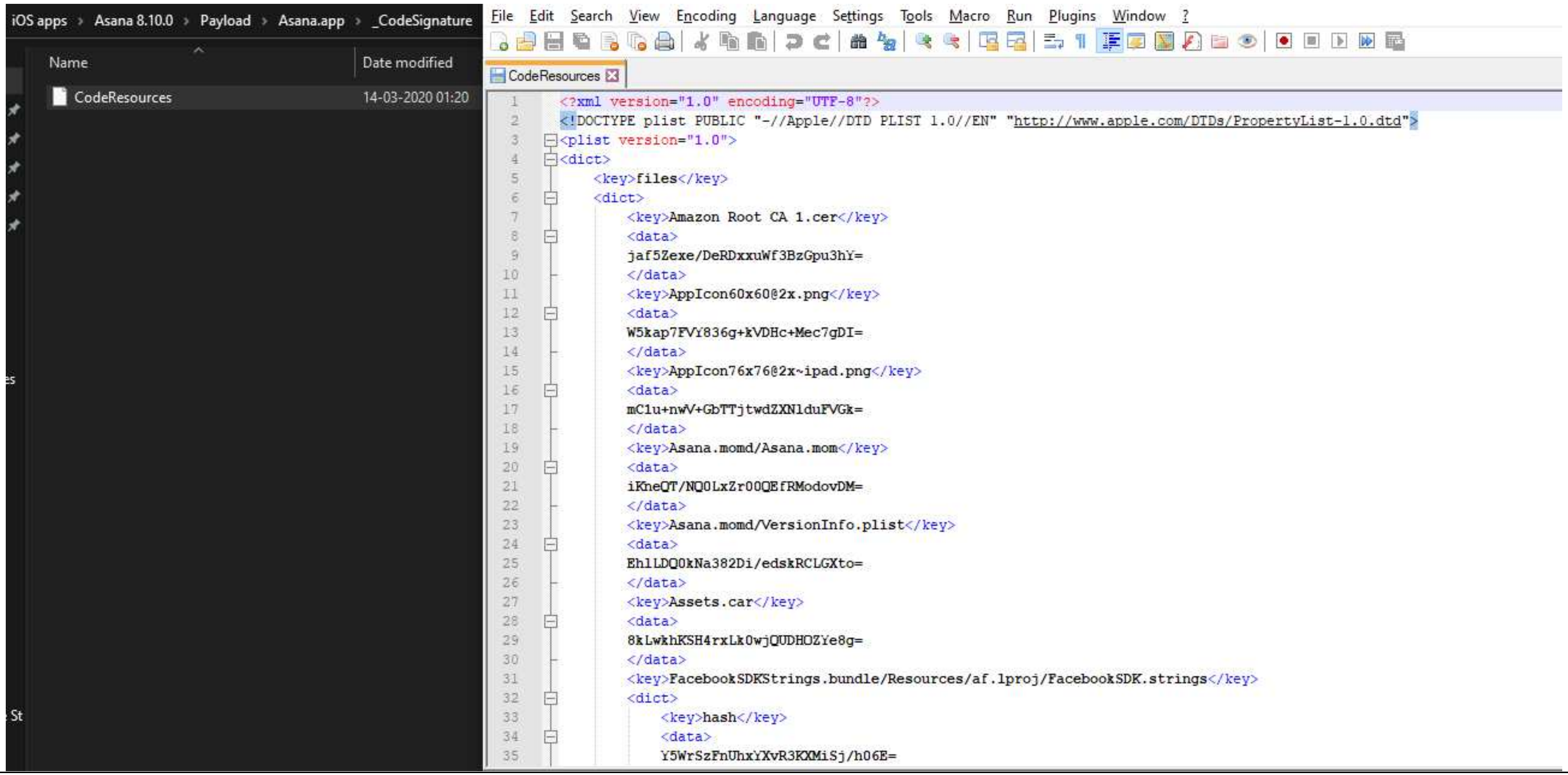
	 pre_master_secret) every time they communicate. To prevent the pre_master_secret from staying in memory any longer than necessary, it should be converted into the master_secret as soon as possible. Client Diffie-Hellman parameters must be compatible with those supplied by the server for the key exchange to work. If the client has a standard DSA or RSA certificate or is unauthenticated, it sends a set of temporary parameters to the server in the client key exchange message, then optionally uses a certificate verify message to authenticate itself. If the same DH keypair is to be used for multiple handshakes, either because the client or server has a certificate containing a fixed DH keypair or because the server is reusing DH keys, care must be taken to prevent small subgroup attacks. Implementations SHOULD follow the guidelines found in [SUBGROUP]. Small subgroup attacks are most easily avoided by using one of the DHE cipher suites and generating a fresh DH private key (X) for each handshake. If a suitable base (such as 2) is chosen, g^X mod p can be computed very quickly; therefore, the performance cost is minimized. Additionally, using a fresh key for each handshake provides Perfect Forward Secrecy. Implementations SHOULD generate a new X for each handshake when using DHE cipher suites. Because TLS allows the server to provide arbitrary DH groups, the client should verify that the DH group is of suitable size as defined by local policy. The client SHOULD also verify that the DH public exponent appears to be of adequate size. [KEYSIZ] provides a useful guide to the strength of various group sizes. The server MAY choose to assist the client by providing a known group, such as those defined in [IKEALG] or [MODP]. These can be verified by simple comparison. Source: The Transport Layer Security (TLS) Protocol Version 1.2, IETF RFC 5246, https://tools.ietf.org/html/rfc5246, Last accessed on Mar 19, 2020 The generated asymmetric key pairs are then used to compute a shared master secret which is then used to encrypt the mobile app download so that it is resistant to observation during transit.

	For **RSA and RSA _PSK**, the RSA public-private key pair is used to encrypt a random premaster secret which is in turn used by Apple server and the user's remote device to compute a master secret. Apple uses the master secret to encrypt the mobile app and the user's remote device uses to decrypt the downloaded mobile app according to the TLS protocol. For **DHE_RSA, ECDHE_RSA, DH_RSA and ECDH_RSA**, Apple server generates an RSA public-private key pair as well as a Diffie-Hellman public-private key pair[21]. The user's remote device also generates a second Diffie-Hellman public-private key pair. Apple server uses its Diffie-Hellman private key and the user's Diffie-Hellman public key to compute a premaster secret, and subsequently compute a master secret. The user's remote device uses the user's Diffie-Hellman private key and Apple's Diffie-Hellman public key to compute the same premaster secret and subsequently the master secret that Apple uses to encrypt the mobile app and the user's remote device uses to decrypt the downloaded mobile app according to the TLS protocol. For **DHE_DSS and DH_DSS**, Apple server generates a DSA public-private key pair as well as a Diffie-Hellman public-private key pair. The user's remote device also generates a second Diffie-Hellman public-private key pair. Apple server uses its Diffie-Hellman private key and the user's Diffie-Hellman public key to compute a premaster secret, and subsequently compute a master secret. The user's remote device uses the user's Diffie-Hellman private key and Apple's Diffie-Hellman public key to compute the same premaster secret and subsequently the master secret that Apple uses to encrypt the mobile app and the user's remote device uses to decrypt the downloaded mobile app according to the TLS protocol. For **ECDH_ECDSA and ECDHE_ECDSA**, Apple server generates an ECDSA public-private key pair as well as a Diffie-Hellman public-private key pair. The user's remote device also generates a second Diffie-Hellman public-private key pair. Apple server uses its Diffie-Hellman private key and the user's Diffie-Hellman public key to compute a premaster secret, and subsequently compute a master secret. The user's remote device uses the user's Diffie-Hellman private key and Apple's Diffie-Hellman public key to compute the same premaster secret and subsequently the master secret that Apple uses to encrypt the mobile app and the user's remote device uses to decrypt the downloaded mobile app according to the TLS protocol. **3. Resistant to Modification Because Mobile App is Code Signed**

[21] ECDH and ECDHE algorithms require generating at the server and the client, elliptical curve parameters that constitute a Diffie-Hellman public-private key pair. See, for example, https://tools.ietf.org/html/rfc5246 page 49-52, https://tools.ietf.org/html/rfc7525 page 12, http://www.cse.hut.fi/fi/opinnot/T-110.5241/2011/luennot-files/Network%20Security%2004%20-%20TLS.pdf, http://csrc.nist.gov/groups/STM/cmvp/documents/140-1/140sp/140sp2897.pdf, http://www.networkworld.com/article/2268575/lan-wan/chapter-2--ssl-vpn-technology.html, http://homes.esat.kuleuven.be/~fvercaut/papers/ACM2012.pdf, Implementing SSL / TLS Using Cryptography and PKI by Joshua Davies, ISBN 1118038770, 9781118038772, Page 305 and Introduction to Computer Networks and Cybersecurity By Chwan-Hwa (John) Wu, J. David Irwin, ISBN 1466572140, 9781466572140, Page 1021, which state that ECDH requires generating a Diffie-Hellman public-private key pair, Last accessed on Mar 19, 2020

	The downloaded mobile app code is resistant to modification, at least in part, because the downloaded app binary is code signed. Code-signing allows users' remote systems to verify that the downloaded app binary is authentic and has not been maliciously modified by a third party. Apple dictates that each developer must sign the mobile app submission with his/her asymmetric developer key that certifies that the app has not been modified by a third party impersonator[22]. *Xcode code signs your app during the build and archive process. If needed, Xcode requests a certificate and adds a signing certificate, the certificate with its public-private key pair, to your keychain. The certificate with the public key is added to your developer account.* Source: https://help.apple.com/xcode/mac/current/#/dev3a05256b8, Last accessed on Mar 19, 2020 *A signing signing certificate includes the certificate with its public-private key pair issued by Apple, and is stored in your keychain. Because the private key is stored locally, protect it as you would an account password. An intermediate certificate is also required to be in your keychain to ensure that your certificate is issued by a certificate authority such as Apple.* *Your signing certificate is added to your keychain and the corresponding certificate is added to your developer account.* *Source:* https://help.apple.com/xcode/mac/current/#/dev1c7c2c67d, Last accessed on Mar 19, 2020 ***About Signing Identities and Certificates*** *Code signing (or signing) an app allows the system to identify who signed the app and to verify that the app has not been modified since it was signed.* *Signing is a requirement for uploading your app to App Store Connect and distributing it through TestFlight or the App Store. The operating system verifies the signature of apps downloaded from the App Store to ensure that apps with invalid signatures don't run. An app's executable code is protected by its signature because the signature becomes invalid if any of the executable code in the app bundle changes. A valid signature lets users trust that the app was signed by an Apple source and hasn't been modified since it was signed.* *Xcode uses your signing certificate to sign your app during the build process. The signing certificate consists of a public-private key pair and a certificate. The private key is used by cryptographic functions to generate the signature. The certificate is issued by Apple; it contains the public key and identifies you as the owner of the key pair. In order to sign apps, you must have both parts of your signing certificate, and an Apple certificate authority in your keychain.*

[22] *See, e.g.,* https://help.apple.com/xcode/mac/current/#/dev3a05256b8, Last accessed on Mar 19, 2020

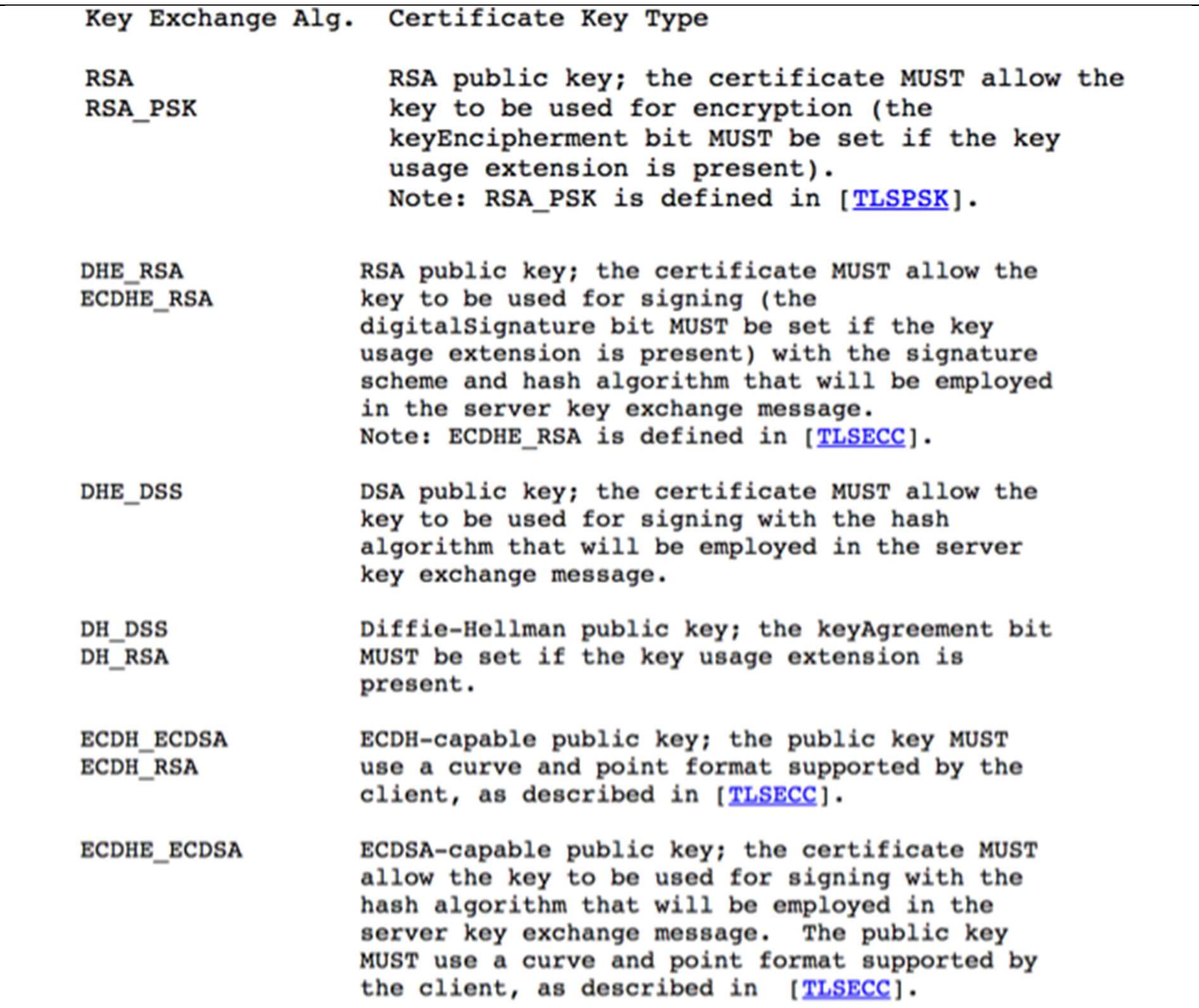
	An app's signature can be removed, and the app can be re-signed using another signing certificate. For example, Apple re-signs all apps sold on the App Store. Also, a fully-tested development build of your app can be re-signed for submission to the App Store. Thus the signature is best understood not as proof of the app's origin but as a verifiable mark placed by the signer. Source: https://help.apple.com/xcode/mac/current/#/devfbe995ebf, Last accessed on Mar 19, 2020 Asana complies with Apple's instructions on code signing as shown by Asana's mobile app contents. Asana's mobile apps contain files such as the file _CodeSignature/CodeResources in Asana's iOS apps which are generated during the code signing process as per instructions from Apple. 

iOS apps › Asana 8.10.0 › Payload › Asana.app › _CodeSignature
Name
Date modified
CodeResources
14-03-2020 01:20
File Edit Search View Encoding Language Settings Tools Macro Run Plugins Window ?
CodeResources

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN" "http://www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
<dict>
    <key>files</key>
    <dict>
        <key>Amazon Root CA 1.cer</key>
        <data>
        jaf5Zexe/DeRDxxuWf3BzGpu3hY=
        </data>
        <key>AppIcon60x60@2x.png</key>
        <data>
        W5kap7FVY836g+kVDHc+Mec7gDI=
        </data>
        <key>AppIcon76x76@2x~ipad.png</key>
        <data>
        mC1u+nwV+GbTTjtwdZXN1duFVGk=
        </data>
        <key>Asana.momd/Asana.mom</key>
        <data>
        iKneQT/NQ0LxZr00QEfRModovDM=
        </data>
        <key>Asana.momd/VersionInfo.plist</key>
        <data>
        Eh1LDQ0kNa382Di/edskRCLGXto=
        </data>
        <key>Assets.car</key>
        <data>
        8kLwkhKSH4rxLk0wjQUDHOZYe8g=
        </data>
        <key>FacebookSDKStrings.bundle/Resources/af.lproj/FacebookSDK.strings</key>
        <dict>
            <key>hash</key>
            <data>
            Y5WrSzFnUhxYXvR3KXMiSj/h06E=
```

	Source: Contents of Asana: organize tasks & work, https://apps.apple.com/us/app/asana-organize-tasks-work/id489969512, as an example of Asana app, Last accessed on Mar 19, 2020 By signing the app binary with a digital signature, Asana's mobile apps are tamper resistant enabling Apple and the iOS mobile devices to verify that the application is being distributed by trusted source (*i.e.* Asana) and that the application has not been modified by a third party, which can be verified by the corresponding public key generated as part of the pair. Thus the app binary is made resistant to modification by digital signing. Accordingly Asana's iOS mobile apps establish SSL/TLS communications with Asana's servers, which involve a SSL/TLS handshake procedure involving asymmetric key encryption. SSL/TLS ensures secure communication and renders the mobile app data further resistant to observation. **4. Resistant to Observation Because Mobile App is Stored on Remote System in Encrypted Form** The mobile app is made further resistant to observation because when downloaded and installed on a user's iOS mobile device, it is stored in an encrypted form. iOS implements disk encryption for encrypting the operating system software, apps and all related data on a mobile device – which further renders Asana app resistant to observation[23]. **5. Resistant to Observation Because Mobile App Securely Communicates with Asana Over SSL/TLS** Asana's mobile apps are made further resistant to observation, at least in part, because the mobile app communicates with Asana using SSL/TLS during operation. Asana app users establish SSL/TLS communications with Asana servers when the app is executed. Such secure communication is necessary to keep source code as well as user identity and activity from being observed in transit from the remote system to Asana servers and vice versa. The secure communications process starts with a TLS handshake procedure which uses asymmetric key encryption and necessitates generation of at least one asymmetric key pair. Specifically, user's remote device negotiates with Asana servers the cipher suite and the key exchange algorithm that will be used for the handshake.

[23] *See, e.g.*, https://www.apple.com/business/docs/iOS_Security_Guide.pdf, Pages 10-18, Last accessed on May 22, 2019



```
Key Exchange Alg.  Certificate Key Type

RSA                RSA public key; the certificate MUST allow the
RSA_PSK            key to be used for encryption (the
                   keyEncipherment bit MUST be set if the key
                   usage extension is present).
                   Note: RSA_PSK is defined in [TLSPSK].

DHE_RSA            RSA public key; the certificate MUST allow the
ECDHE_RSA          key to be used for signing (the
                   digitalSignature bit MUST be set if the key
                   usage extension is present) with the signature
                   scheme and hash algorithm that will be employed
                   in the server key exchange message.
                   Note: ECDHE_RSA is defined in [TLSECC].

DHE_DSS            DSA public key; the certificate MUST allow the
                   key to be used for signing with the hash
                   algorithm that will be employed in the server
                   key exchange message.

DH_DSS             Diffie-Hellman public key; the keyAgreement bit
DH_RSA             MUST be set if the key usage extension is
                   present.

ECDH_ECDSA         ECDH-capable public key; the public key MUST
ECDH_RSA           use a curve and point format supported by the
                   client, as described in [TLSECC].

ECDHE_ECDSA        ECDSA-capable public key; the certificate MUST
                   allow the key to be used for signing with the
                   hash algorithm that will be employed in the
                   server key exchange message.  The public key
                   MUST use a curve and point format supported by
                   the client, as described in  [TLSECC].
```

Source: https://tools.ietf.org/html/rfc5246 at 48-49, Last accessed on Mar 19, 2020

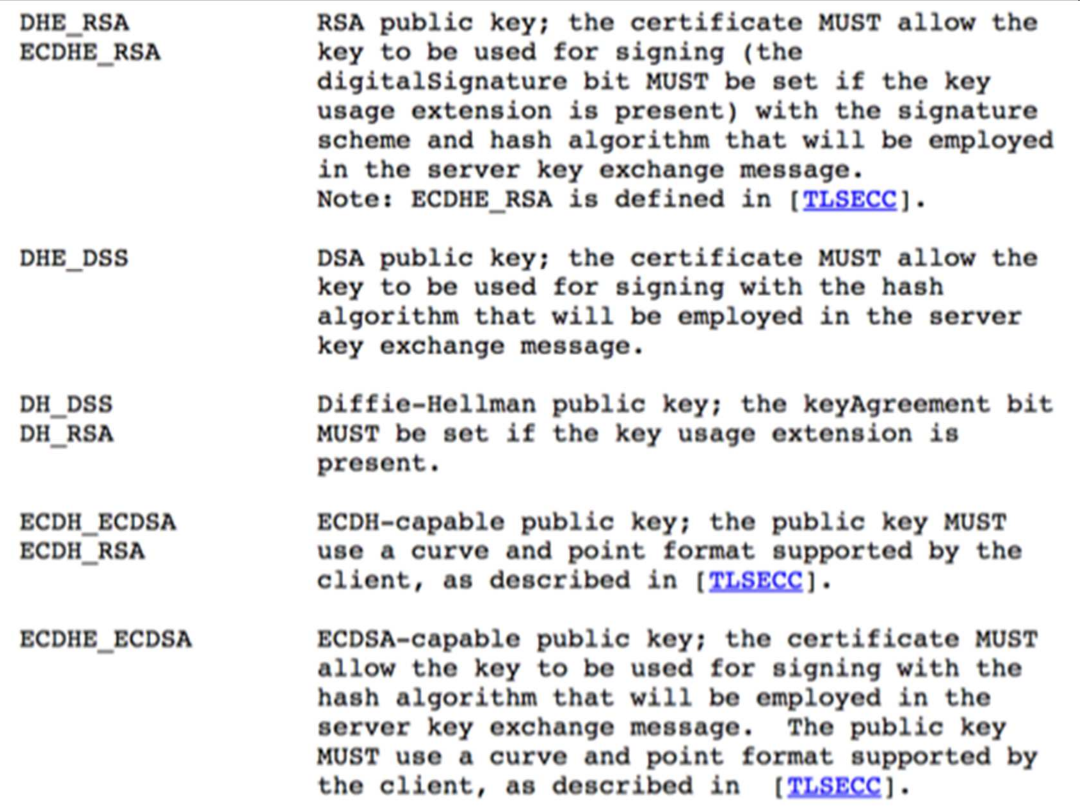
Each of these algorithms necessitates generating one or more asymmetric key pairs – that are in turn used to compute a shared master secret for encrypting communication between Asana and user's remote device.

For **RSA and RSA _PSK**, Asana server generates an RSA public-private key pair.

	For **DHE_RSA, ECDHE_RSA, DH_RSA and ECDH_RSA**, Asana server generates an RSA public-private key pair as well as a Diffie-Hellman public-private key pair[24]. The user's remote device also generates a second Diffie-Hellman public-private key pair. For **DHE_DSS and DH_DSS**, Asana server generates a DSA public-private key pair as well as a Diffie-Hellman public-private key pair. The user's remote device also generates a second Diffie-Hellman public-private key pair. For **ECDH_ECDSA and ECDHE_ECDSA**, Asana server generates an ECDSA public-private key pair as well as a Diffie-Hellman public-private key pair. The user's remote device also generates a second Diffie-Hellman public-private key pair.

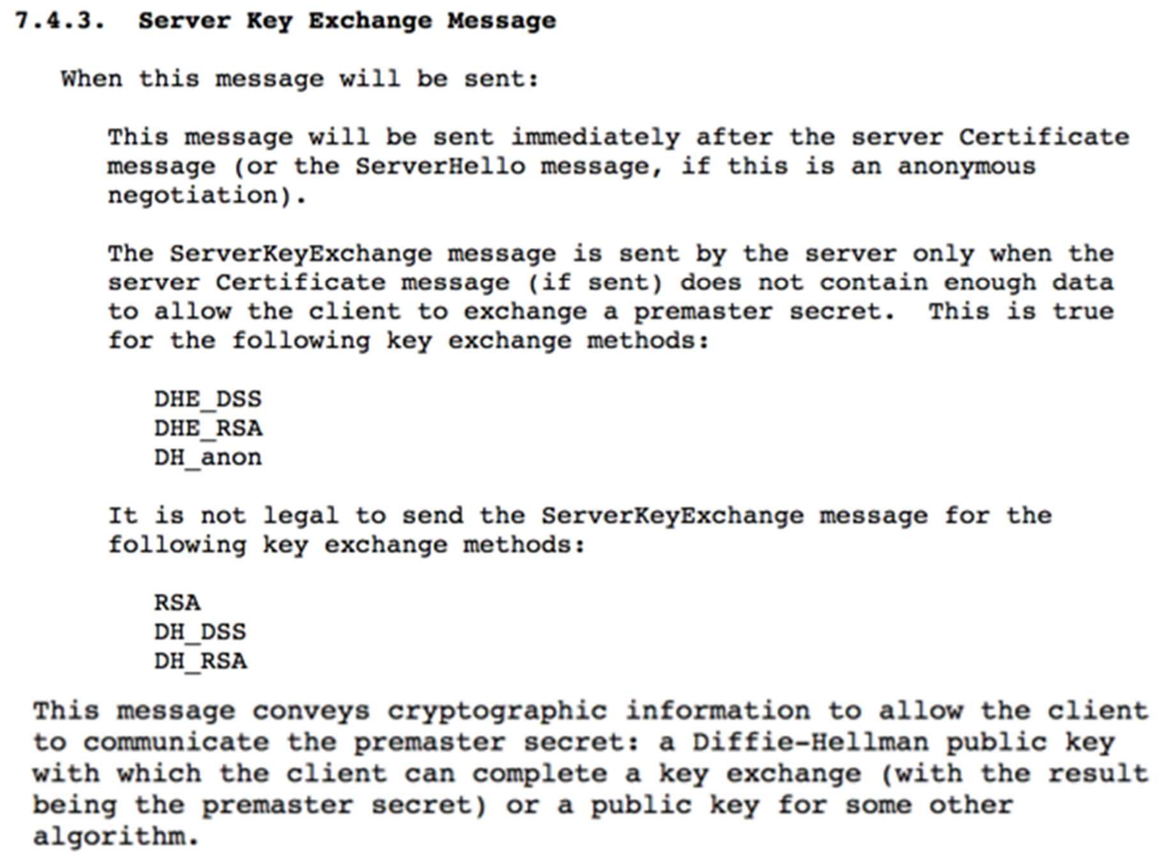
[24] ECDH and ECDHE algorithms require generating at the server and the client, elliptical curve parameters that constitute a Diffie-Hellman public-private key pair. See, for example, https://tools.ietf.org/html/rfc5246 page 49-52, https://tools.ietf.org/html/rfc7525 page 12, http://www.cse.hut.fi/fi/opinnot/T-110.5241/2011/luennot-files/Network%20Security%2004%20-%20TLS.pdf, http://csrc.nist.gov/groups/STM/cmvp/documents/140-1/140sp/140sp2897.pdf, http://www.networkworld.com/article/2268575/lan-wan/chapter-2--ssl-vpn-technology.html, , Last accessed on Mar 19, 2020

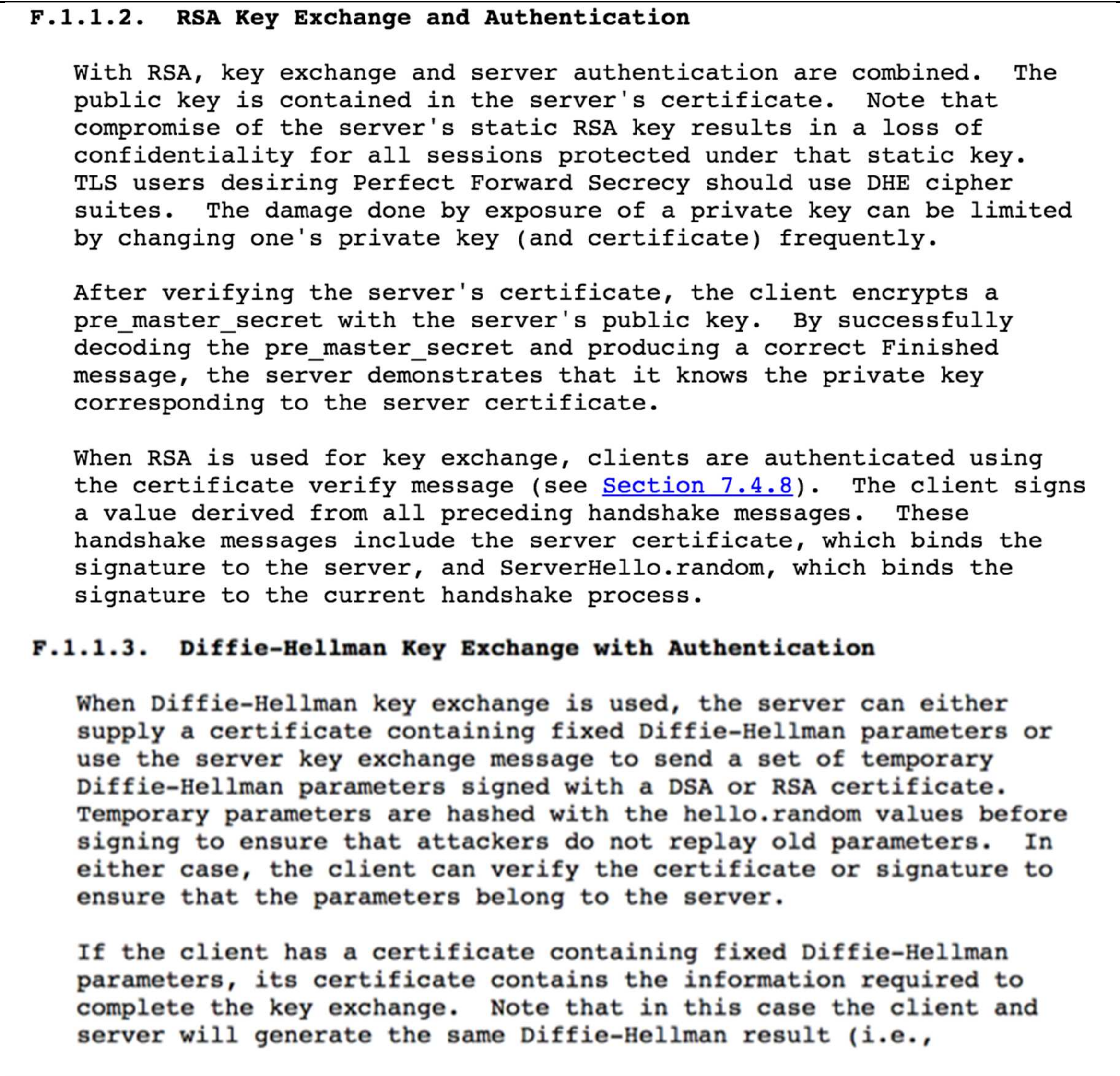
http://homes.esat.kuleuven.be/~fvercaut/papers/ACM2012.pdf, Implementing SSL / TLS Using Cryptography and PKI by Joshua Davies, ISBN 1118038770, 9781118038772, Page 305 and Introduction to Computer Networks and Cybersecurity By Chwan-Hwa (John) Wu, J. David Irwin, ISBN
1466572140, 9781466572140, Page 1021, which state that ECDH requires generating a Diffie-Hellman public-private key pair, Last accessed on Mar 19, 2020



```
DHE_RSA            RSA public key; the certificate MUST allow the
ECDHE_RSA          key to be used for signing (the
                   digitalSignature bit MUST be set if the key
                   usage extension is present) with the signature
                   scheme and hash algorithm that will be employed
                   in the server key exchange message.
                   Note: ECDHE_RSA is defined in [TLSECC].

DHE_DSS            DSA public key; the certificate MUST allow the
                   key to be used for signing with the hash
                   algorithm that will be employed in the server
                   key exchange message.

DH_DSS             Diffie-Hellman public key; the keyAgreement bit
DH_RSA             MUST be set if the key usage extension is
                   present.

ECDH_ECDSA         ECDH-capable public key; the public key MUST
ECDH_RSA           use a curve and point format supported by the
                   client, as described in [TLSECC].

ECDHE_ECDSA        ECDSA-capable public key; the certificate MUST
                   allow the key to be used for signing with the
                   hash algorithm that will be employed in the
                   server key exchange message.  The public key
                   MUST use a curve and point format supported by
                   the client, as described in  [TLSECC].
```

Source: https://tools.ietf.org/html/rfc5246 at 48-49, Last accessed on Mar 19, 2020

	 **7.4.3. Server Key Exchange Message** When this message will be sent: This message will be sent immediately after the server Certificate message (or the ServerHello message, if this is an anonymous negotiation). The ServerKeyExchange message is sent by the server only when the server Certificate message (if sent) does not contain enough data to allow the client to exchange a premaster secret. This is true for the following key exchange methods: DHE_DSS DHE_RSA DH_anon It is not legal to send the ServerKeyExchange message for the following key exchange methods: RSA DH_DSS DH_RSA This message conveys cryptographic information to allow the client to communicate the premaster secret: a Diffie-Hellman public key with which the client can complete a key exchange (with the result being the premaster secret) or a public key for some other algorithm. *Source: https://tools.ietf.org/html/rfc5246 at 50-51*, Last accessed on Mar 19, 2020



F.1.1.2. RSA Key Exchange and Authentication

With RSA, key exchange and server authentication are combined. The public key is contained in the server's certificate. Note that compromise of the server's static RSA key results in a loss of confidentiality for all sessions protected under that static key. TLS users desiring Perfect Forward Secrecy should use DHE cipher suites. The damage done by exposure of a private key can be limited by changing one's private key (and certificate) frequently.

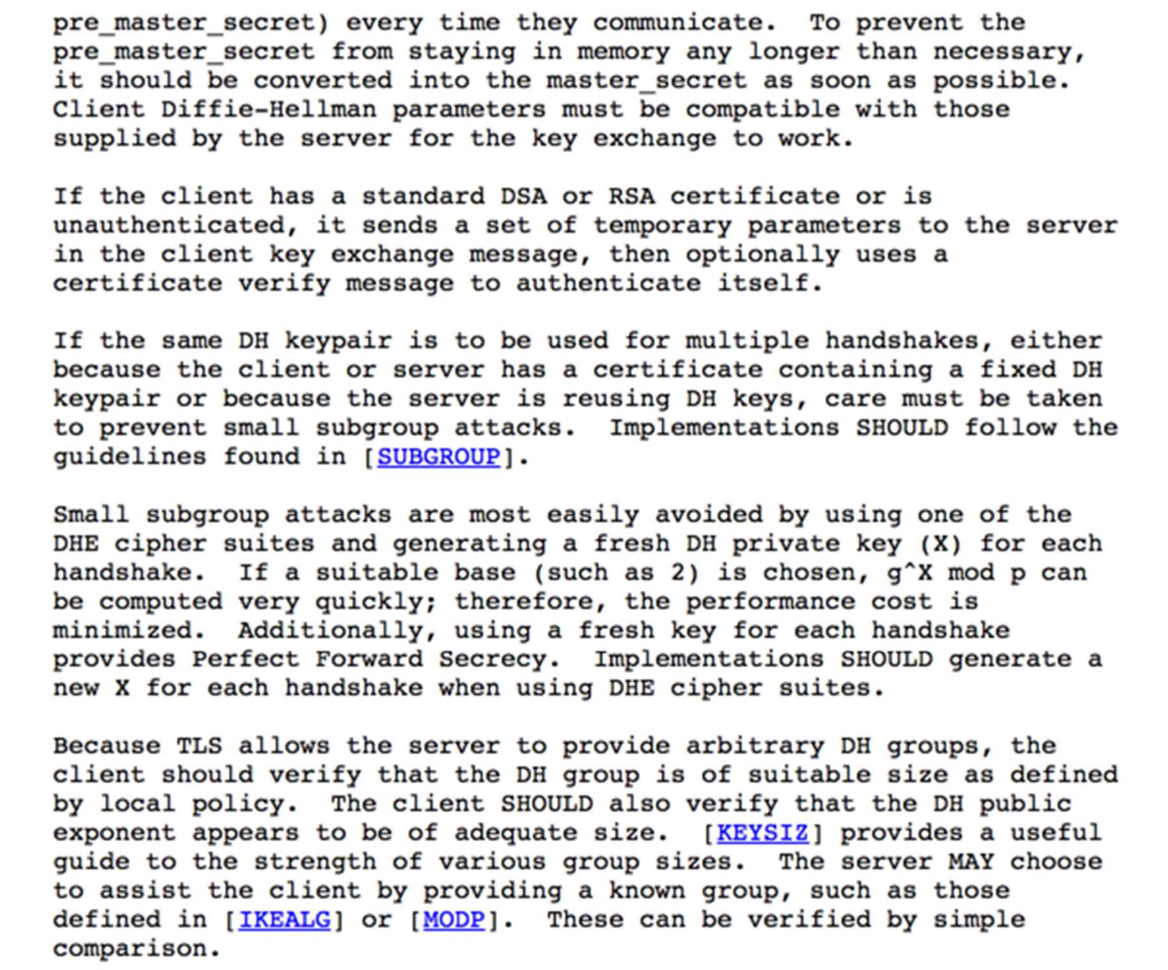
After verifying the server's certificate, the client encrypts a pre_master_secret with the server's public key. By successfully decoding the pre_master_secret and producing a correct Finished message, the server demonstrates that it knows the private key corresponding to the server certificate.

When RSA is used for key exchange, clients are authenticated using the certificate verify message (see Section 7.4.8). The client signs a value derived from all preceding handshake messages. These handshake messages include the server certificate, which binds the signature to the server, and ServerHello.random, which binds the signature to the current handshake process.

F.1.1.3. Diffie-Hellman Key Exchange with Authentication

When Diffie-Hellman key exchange is used, the server can either supply a certificate containing fixed Diffie-Hellman parameters or use the server key exchange message to send a set of temporary Diffie-Hellman parameters signed with a DSA or RSA certificate. Temporary parameters are hashed with the hello.random values before signing to ensure that attackers do not replay old parameters. In either case, the client can verify the certificate or signature to ensure that the parameters belong to the server.

If the client has a certificate containing fixed Diffie-Hellman parameters, its certificate contains the information required to complete the key exchange. Note that in this case the client and server will generate the same Diffie-Hellman result (i.e.,



pre_master_secret) every time they communicate. To prevent the pre_master_secret from staying in memory any longer than necessary, it should be converted into the master_secret as soon as possible. Client Diffie-Hellman parameters must be compatible with those supplied by the server for the key exchange to work.

If the client has a standard DSA or RSA certificate or is unauthenticated, it sends a set of temporary parameters to the server in the client key exchange message, then optionally uses a certificate verify message to authenticate itself.

If the same DH keypair is to be used for multiple handshakes, either because the client or server has a certificate containing a fixed DH keypair or because the server is reusing DH keys, care must be taken to prevent small subgroup attacks. Implementations SHOULD follow the guidelines found in [SUBGROUP].

Small subgroup attacks are most easily avoided by using one of the DHE cipher suites and generating a fresh DH private key (X) for each handshake. If a suitable base (such as 2) is chosen, g^X mod p can be computed very quickly; therefore, the performance cost is minimized. Additionally, using a fresh key for each handshake provides Perfect Forward Secrecy. Implementations SHOULD generate a new X for each handshake when using DHE cipher suites.

Because TLS allows the server to provide arbitrary DH groups, the client should verify that the DH group is of suitable size as defined by local policy. The client SHOULD also verify that the DH public exponent appears to be of adequate size. [KEYSIZ] provides a useful guide to the strength of various group sizes. The server MAY choose to assist the client by providing a known group, such as those defined in [IKEALG] or [MODP]. These can be verified by simple comparison.

Source: The Transport Layer Security (TLS) Protocol Version 1.2, IETF RFC 5246, https://tools.ietf.org/html/rfc5246, Last accessed on Mar 19, 2020

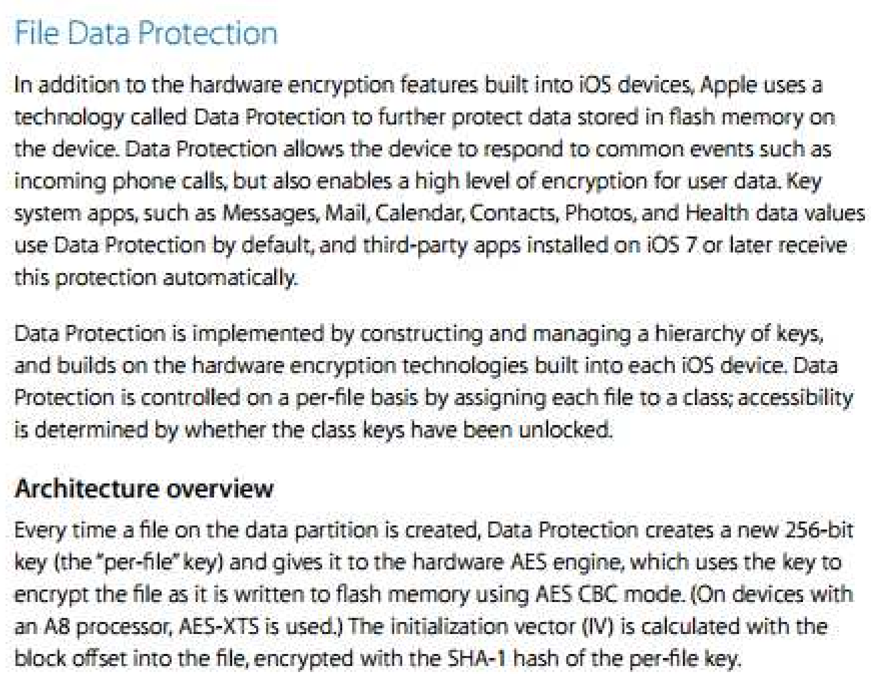
The generated asymmetric key pairs are then used to compute a shared master secret which is then used to encrypt subsequent communications between Asana and the user's remote device so that they are resistant to observation during transit.

	For **RSA and RSA _PSK**, the RSA public-private key pair is used to encrypt a random premaster secret which is in turn used by Asana server and the user's remote device to compute a master secret. Asana and the user's remote device use the master secret for encrypting and decrypting communication messages. For **DHE_RSA, ECDHE_RSA, DH_RSA and ECDH_RSA**, Asana server generates an RSA public-private key pair as well as a Diffie-Hellman public-private key pair[25]. The user's remote device also generates a second Diffie-Hellman public-private key pair. Asana server uses its Diffie-Hellman private key and the user's Diffie-Hellman public key to compute a premaster secret, and subsequently compute a master secret. The user's remote device uses the user's Diffie-Hellman private key and Apple's Diffie-Hellman public key to compute the same premaster secret and subsequently the master secret for encrypting and decrypting communication messages. For **DHE_DSS and DH_DSS**, Asana server generates a DSA public-private key pair as well as a Diffie-Hellman public-private key pair. The user's remote device also generates a second Diffie-Hellman public-private key pair. Asana server uses its Diffie-Hellman private key and the user's Diffie-Hellman public key to compute a premaster secret, and subsequently compute a master secret. The user's remote device uses the user's Diffie-Hellman private key and Apple's Diffie-Hellman public key to compute the same premaster secret and subsequently the master secret for encrypting and decrypting communication messages. For **ECDH_ECDSA and ECDHE_ECDSA**, Asana server generates an ECDSA public-private key pair as well as a Diffie-Hellman public-private key pair. The user's remote device also generates a second Diffie-Hellman public-private key pair. Asana server uses its Diffie-Hellman private key and the user's Diffie-Hellman public key to compute a premaster secret, and subsequently compute a master secret. The user's remote device uses the user's Diffie-Hellman private key and Apple's Diffie-Hellman public key to compute the same premaster secret and subsequently the master secret for encrypting and decrypting communication messages.
9. The method of claim 1, wherein building the executable tamper resistant code module comprises generating an integrity verification kernel.	Asana's mobile software application products and services including by way of example, but not limited to the following apps ("mobile applications", "mobile apps" or "Accused Products") that are specifically developed, used, sold, offered for sale, marketed, licensed and distributed by Asana to be downloaded onto Apple iOS mobile or tablet devices.

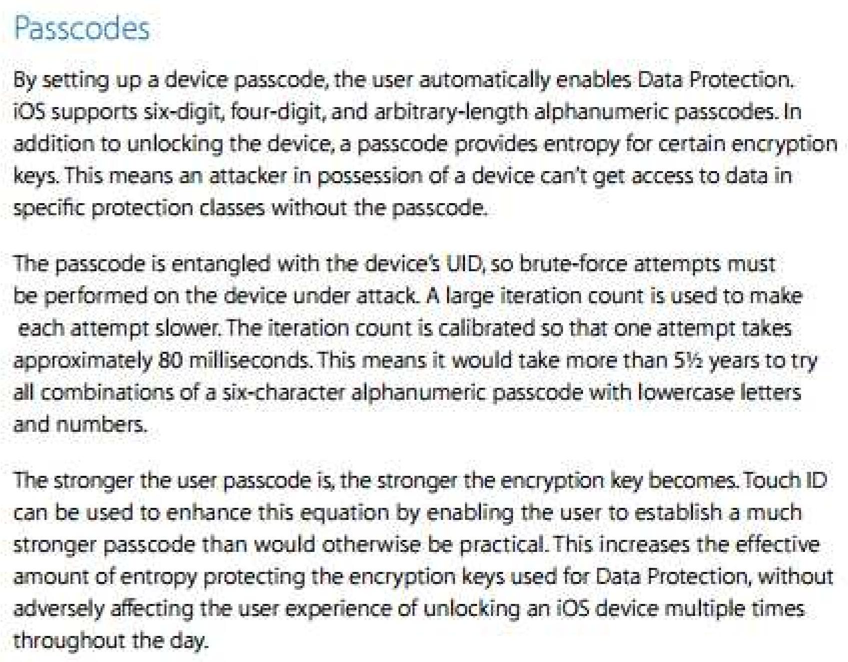
[25] ECDH and ECDHE algorithms require generating at the server and the client, elliptical curve parameters that constitute a Diffie-Hellman public-private key pair. See, for example, https://tools.ietf.org/html/rfc5246 page 49-52, https://tools.ietf.org/html/rfc7525 page 12, http://www.cse.hut.fi/fi/opinnot/T-110.5241/2011/luennot-files/Network%20Security%2004%20-%20TLS.pdf, http://csrc.nist.gov/groups/STM/cmvp/documents/140-1/140sp/140sp2897.pdf, http://www.networkworld.com/article/2268575/lan-wan/chapter-2--ssl-vpn-technology.html, http://homes.esat.kuleuven.be/~fvercaut/papers/ACM2012.pdf, Implementing SSL / TLS Using Cryptography and PKI by Joshua Davies, ISBN 1118038770, 9781118038772, Page 305 and Introduction to Computer Networks and Cybersecurity By Chwan-Hwa (John) Wu, J. David Irwin, ISBN 1466572140, 9781466572140, Page 1021, which state that ECDH requires generating a Diffie-Hellman public-private key pair, Last accessed on Mar 19, 2020

	• Asana: organize tasks & work (https://apps.apple.com/us/app/asana-organize-tasks-work/id489969512), Last accessed on Mar 19, 2020 Asana directly infringes and/or continues to knowingly induce Apple to infringe this claim by intentionally developing, making, marketing, advertising, providing, sending, distributing and licensing its mobile applications software, documentation, materials, training or support and aiding, abetting, encouraging, promoting or inviting use thereof. Upon information and belief, the method step of generating an integrity verification kernel is performed by Apple and/or its agents – whose acts are attributable to Asana (i) because Asana works together with Apple in a joint enterprise in the building and distribution of its mobile apps, or (ii) because Apple distributes and markets Asana's mobile apps under the direction and control of Asana, or acts as agent, or on behalf of Asana, in the building, marketing and distribution of Asana's mobile apps. Alternatively, to the extent any portion of this method step is performed by Asana, such acts are attributable to Apple, who conditions participation in and the receipt of a benefit, namely, the distribution of Asana's mobile apps through its app store, upon compliance with certain mandatory procedures and guidelines dictated by Apple in the building and upload of Asana's mobile apps, and Asana induces infringement by Apple in the building, marketing and distribution of Asana's mobile apps. Alternatively, any steps or acts performed by Asana, are attributable to Apple, who conditions participation in and the receipt of a benefit, namely, the distribution of Asana's mobile apps through its app store, upon compliance with certain mandatory procedures and guidelines dictated by Apple in the building and upload of Asana's mobile apps, and Asana induces infringement by Apple in the building, marketing and distribution of Asana's mobile apps. The Court construed "integrity verification kernel" to mean "software that verifies that a program image corresponds to a supplied digital signature and that is resistant to observation and modification." When a mobile app is deployed on device, a hash algorithm is used to compute hashes for the resources. The public key is used to decrypt the hashes in the hash manifest. Then, the hashes from the hash manifest are compared with the previous hash computations. If the hashes do not match, the digital signature is invalid and the application does not launch. Thus, the private key described above is used to digitally sign the program image for the app, and this digital signature is supplied so that the mobile app can be verified prior to launch of the mobile application[26]. The code for iOS used to validate the digitally signed mobile app code is itself resistant to observation and modification.

[26] *See, e.g.*, https://developer.apple.com/library/archive/documentation/Security/Conceptual/CodeSigningGuide/AboutCS/AboutCS.html, Last accessed on Mar 19, 2020

	Further, iOS implements disk encryption for encrypting the operating system software, applications and all related data on a mobile device – which renders the code for validating the digitally signed mobile app binary further resistant to observation[27]. Further iOS implements a secure boot functionality that verifies the operating system, including code for validating digitally signed mobile apps, when the mobile device is powered on. If this verification fails, *i.e.* if the operating system has been maliciously modified, the operating system does not launch[28]. Thus, the software that verifies that a program image corresponds to a supplied digital signature is both resistant to observation and modification.  File Data Protection In addition to the hardware encryption features built into iOS devices, Apple uses a technology called Data Protection to further protect data stored in flash memory on the device. Data Protection allows the device to respond to common events such as incoming phone calls, but also enables a high level of encryption for user data. Key system apps, such as Messages, Mail, Calendar, Contacts, Photos, and Health data values use Data Protection by default, and third-party apps installed on iOS 7 or later receive this protection automatically. Data Protection is implemented by constructing and managing a hierarchy of keys, and builds on the hardware encryption technologies built into each iOS device. Data Protection is controlled on a per-file basis by assigning each file to a class; accessibility is determined by whether the class keys have been unlocked. **Architecture overview** Every time a file on the data partition is created, Data Protection creates a new 256-bit key (the "per-file" key) and gives it to the hardware AES engine, which uses the key to encrypt the file as it is written to flash memory using AES CBC mode. (On devices with an A8 processor, AES-XTS is used.) The initialization vector (IV) is calculated with the block offset into the file, encrypted with the SHA-1 hash of the per-file key. Source: iOS Security Guide, https://www.apple.com/business/docs/iOS_Security_Guide.pdf, Last accessed on May 22, 2019

[27] *See, e.g.,* https://www.apple.com/business/docs/iOS_Security_Guide.pdf, Pages 10-18, Last accessed on May 22, 2019
[28] *See, e.g.,* https://www.apple.com/business/docs/iOS_Security_Guide.pdf, Pages 5-6, Last accessed on May 22, 2019



Passcodes

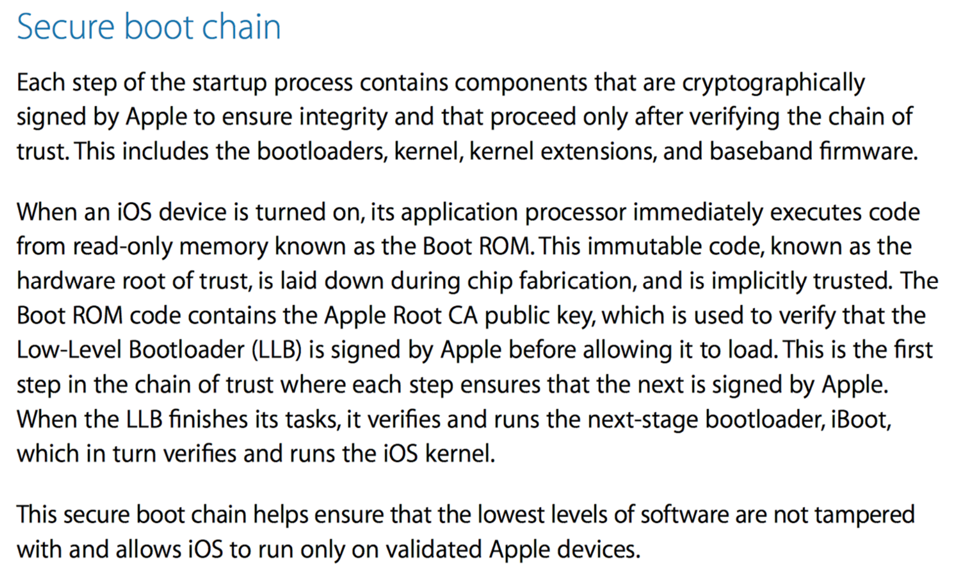
By setting up a device passcode, the user automatically enables Data Protection. iOS supports six-digit, four-digit, and arbitrary-length alphanumeric passcodes. In addition to unlocking the device, a passcode provides entropy for certain encryption keys. This means an attacker in possession of a device can't get access to data in specific protection classes without the passcode.

The passcode is entangled with the device's UID, so brute-force attempts must be performed on the device under attack. A large iteration count is used to make each attempt slower. The iteration count is calibrated so that one attempt takes approximately 80 milliseconds. This means it would take more than 5½ years to try all combinations of a six-character alphanumeric passcode with lowercase letters and numbers.

The stronger the user passcode is, the stronger the encryption key becomes. Touch ID can be used to enhance this equation by enabling the user to establish a much stronger passcode than would otherwise be practical. This increases the effective amount of entropy protecting the encryption keys used for Data Protection, without adversely affecting the user experience of unlocking an iOS device multiple times throughout the day.

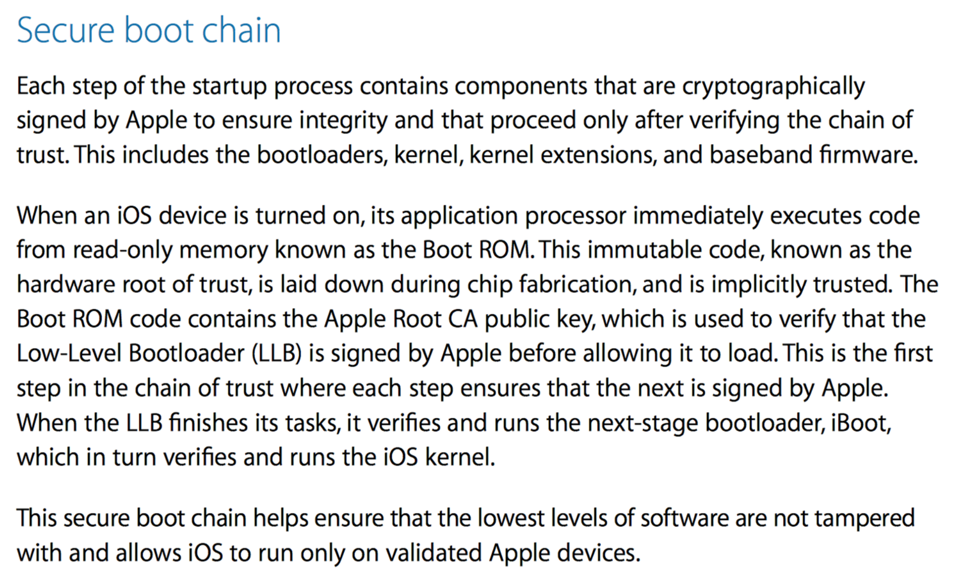
Source: iOS Security Guide, https://www.apple.com/business/docs/iOS_Security_Guide.pdf, Last accessed on May 22, 2019

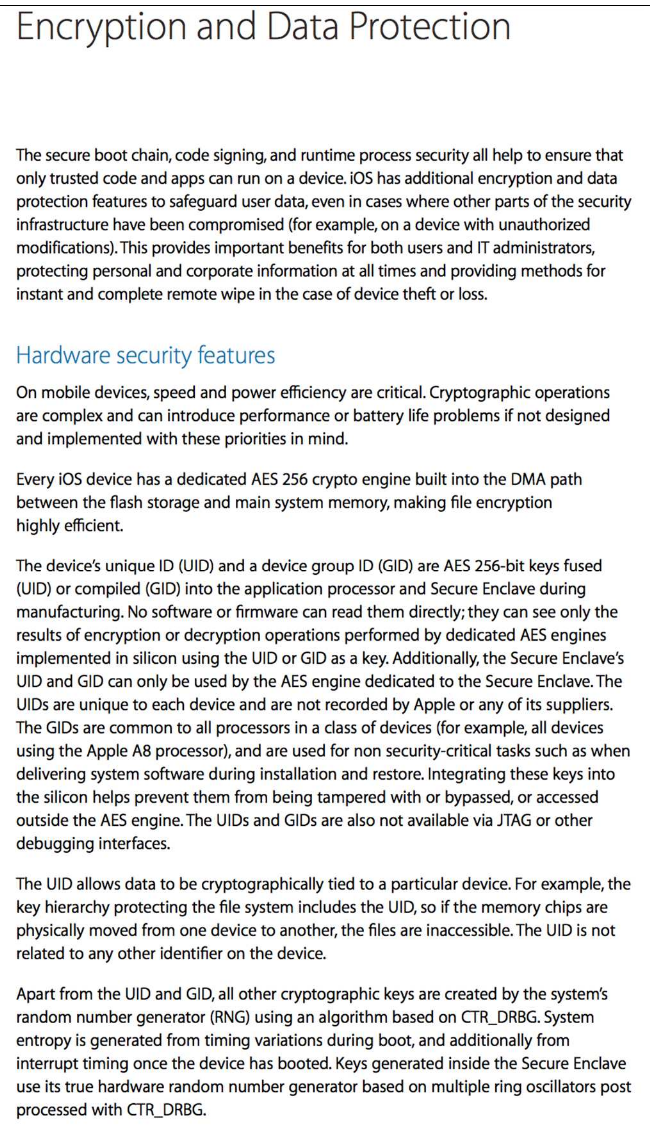
Further iOS implements a secure boot functionality that verifies the operating system, including code for validating digitally signed mobile apps, when the mobile device is powered on. If this verification fails, i.e. if the operating system has been maliciously modified, the operating system does not launch.

	 Secure boot chain Each step of the startup process contains components that are cryptographically signed by Apple to ensure integrity and that proceed only after verifying the chain of trust. This includes the bootloaders, kernel, kernel extensions, and baseband firmware. When an iOS device is turned on, its application processor immediately executes code from read-only memory known as the Boot ROM. This immutable code, known as the hardware root of trust, is laid down during chip fabrication, and is implicitly trusted. The Boot ROM code contains the Apple Root CA public key, which is used to verify that the Low-Level Bootloader (LLB) is signed by Apple before allowing it to load. This is the first step in the chain of trust where each step ensures that the next is signed by Apple. When the LLB finishes its tasks, it verifies and runs the next-stage bootloader, iBoot, which in turn verifies and runs the iOS kernel. This secure boot chain helps ensure that the lowest levels of software are not tampered with and allows iOS to run only on validated Apple devices. Source: https://www.apple.com/business/docs/iOS_Security_Guide.pdf, Last accessed on May 22, 2019
10. The method of claim 9, wherein generating an integrity verification kernel comprises accessing an asymmetric public key of a predetermined asymmetric key pair associated with a manifest of the program signed by an asymmetric private key of the predetermined asymmetric key pair, producing integrity verification kernel code with the asymmetric public key for verifying the signed manifest of the program and combining manifest parser generator code and the integrity verification kernel code to	Asana's mobile software application products and services including by way of example, but not limited to the following apps ("mobile applications", "mobile apps" or "Accused Products") that are specifically developed, used, sold, offered for sale, marketed, licensed and distributed by Asana to be downloaded onto Apple iOS mobile or tablet devices. • Asana: organize tasks & work (https://apps.apple.com/us/app/asana-organize-tasks-work/id489969512), Last accessed on Mar 19, 2020 Asana directly infringes and/or continues to knowingly induce Apple to infringe this claim by intentionally developing, making, marketing, advertising, providing, sending, distributing and licensing its mobile applications software, documentation, materials, training or support and aiding, abetting, encouraging, promoting or inviting use thereof. Upon information and belief, the method step of accessing an asymmetric public key of a predetermined asymmetric key pair associated with a manifest of the program signed by an asymmetric private key of the predetermined asymmetric key pair, producing integrity verification kernel code with the asymmetric public key for verifying the signed manifest of the program and combining manifest parser generator code and the integrity verification kernel code to produce the integrity verification kernel is performed by Apple and/or its agents – whose acts are attributable to Asana (i) because Asana works together with Apple in a joint enterprise in the building and distribution of its mobile apps, or (ii) because Apple distributes and

produce the integrity verification kernel.	markets Asana's mobile apps under the direction and control of Asana, or acts as agent, or on behalf of Asana, in the building, marketing and distribution of Asana's mobile apps. Alternatively, to the extent any portion of this method step is performed by Asana, such acts are attributable to Apple, who conditions participation in and the receipt of a benefit, namely, the distribution of Asana's mobile apps through its app store, upon compliance with certain mandatory procedures and guidelines dictated by Apple in the building and upload of Asana's mobile apps, and Asana induces infringement by Apple in the building, marketing and distribution of Asana's mobile apps. The Court previously construed[29] "integrity verification kernel" to mean "software that verifies that a program image corresponds to a supplied digital signature and that is resistant to observation and modification" and "manifest" to mean ""static source code that includes the integrity verification kernel's entry code, generator code, accumulator code, and other code for tamper detection." iOS implements a secure boot functionality that verifies the operating system, including code for validating digitally signed mobile apps, when the mobile device is powered on. If this verification fails, i.e. if the operating system has been maliciously modified, the operating system, along with the mobile apps installed on the device, does not launch. Thus every time the device is restarted an integrity verification kernel is generated by accessing an asymmetric public key of a predetermined asymmetric key pair associated with a manifest of the operating system signed with the corresponding asymmetric private key by Apple. This integrity verification kernel is implemented in code which is produced by combining code that parses a manifest associated with the operating system and code that verifies that the operating system on the device matches with the manifest and has not been maliciously modified. Further, the software that performs the above verification is stored in a compiled and encrypted form and is hence resistant to observation. Additionally, the software is digitally signed by Apple with their asymmetric private key and is thus resistant to modification.

[29] See Memorandum Opinion and Order, Document 104 signed by Judge Rodney Gilstrap on 7/21/2016 in re Plano Encryption Technologies, LLC v. American Bank of Texas (2:15-cv-01273).

	 Secure boot chain Each step of the startup process contains components that are cryptographically signed by Apple to ensure integrity and that proceed only after verifying the chain of trust. This includes the bootloaders, kernel, kernel extensions, and baseband firmware. When an iOS device is turned on, its application processor immediately executes code from read-only memory known as the Boot ROM. This immutable code, known as the hardware root of trust, is laid down during chip fabrication, and is implicitly trusted. The Boot ROM code contains the Apple Root CA public key, which is used to verify that the Low-Level Bootloader (LLB) is signed by Apple before allowing it to load. This is the first step in the chain of trust where each step ensures that the next is signed by Apple. When the LLB finishes its tasks, it verifies and runs the next-stage bootloader, iBoot, which in turn verifies and runs the iOS kernel. This secure boot chain helps ensure that the lowest levels of software are not tampered with and allows iOS to run only on validated Apple devices. Source: https://www.apple.com/business/docs/iOS_Security_Guide.pdf, Last accessed on May 22, 2019



Encryption and Data Protection

The secure boot chain, code signing, and runtime process security all help to ensure that only trusted code and apps can run on a device. iOS has additional encryption and data protection features to safeguard user data, even in cases where other parts of the security infrastructure have been compromised (for example, on a device with unauthorized modifications). This provides important benefits for both users and IT administrators, protecting personal and corporate information at all times and providing methods for instant and complete remote wipe in the case of device theft or loss.

Hardware security features

On mobile devices, speed and power efficiency are critical. Cryptographic operations are complex and can introduce performance or battery life problems if not designed and implemented with these priorities in mind.

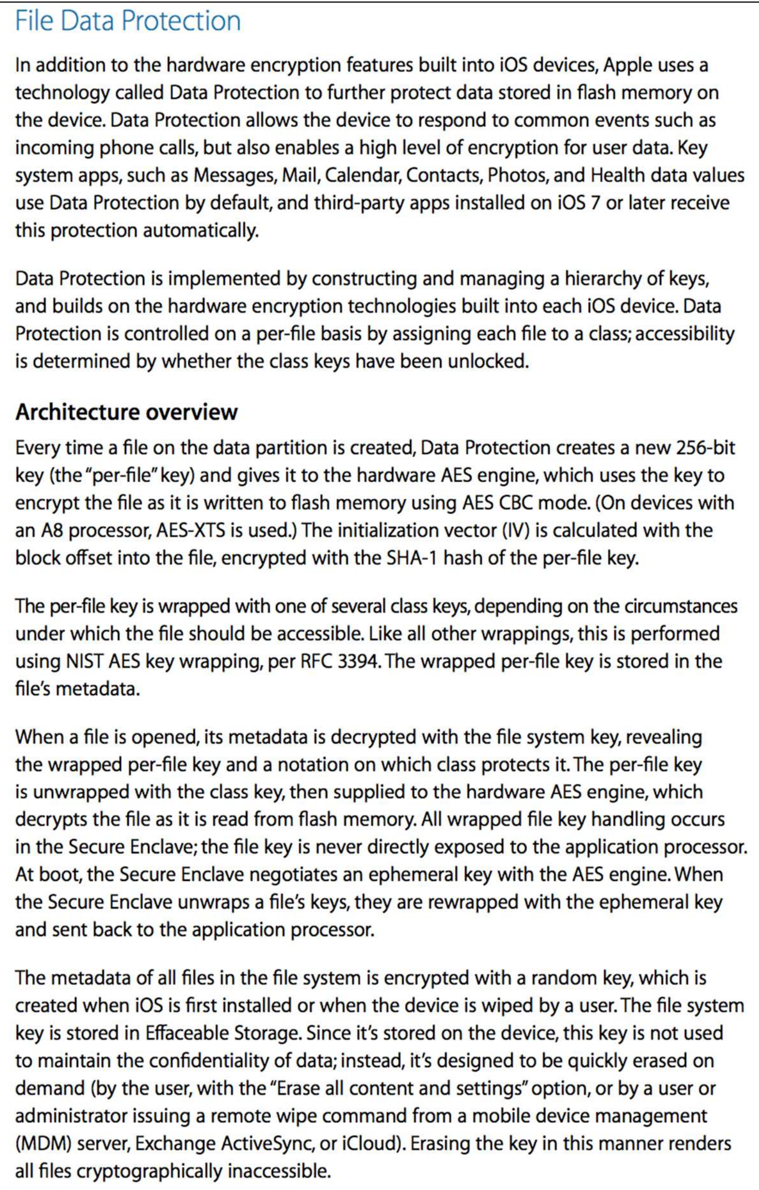
Every iOS device has a dedicated AES 256 crypto engine built into the DMA path between the flash storage and main system memory, making file encryption highly efficient.

The device's unique ID (UID) and a device group ID (GID) are AES 256-bit keys fused (UID) or compiled (GID) into the application processor and Secure Enclave during manufacturing. No software or firmware can read them directly; they can see only the results of encryption or decryption operations performed by dedicated AES engines implemented in silicon using the UID or GID as a key. Additionally, the Secure Enclave's UID and GID can only be used by the AES engine dedicated to the Secure Enclave. The UIDs are unique to each device and are not recorded by Apple or any of its suppliers. The GIDs are common to all processors in a class of devices (for example, all devices using the Apple A8 processor), and are used for non security-critical tasks such as when delivering system software during installation and restore. Integrating these keys into the silicon helps prevent them from being tampered with or bypassed, or accessed outside the AES engine. The UIDs and GIDs are also not available via JTAG or other debugging interfaces.

The UID allows data to be cryptographically tied to a particular device. For example, the key hierarchy protecting the file system includes the UID, so if the memory chips are physically moved from one device to another, the files are inaccessible. The UID is not related to any other identifier on the device.

Apart from the UID and GID, all other cryptographic keys are created by the system's random number generator (RNG) using an algorithm based on CTR_DRBG. System entropy is generated from timing variations during boot, and additionally from interrupt timing once the device has booted. Keys generated inside the Secure Enclave use its true hardware random number generator based on multiple ring oscillators post processed with CTR_DRBG.

Source: iOS Security Guide, Page 10, https://www.apple.com/business/docs/iOS_Security_Guide.pdf, Last accessed on May 22, 2019



File Data Protection

In addition to the hardware encryption features built into iOS devices, Apple uses a technology called Data Protection to further protect data stored in flash memory on the device. Data Protection allows the device to respond to common events such as incoming phone calls, but also enables a high level of encryption for user data. Key system apps, such as Messages, Mail, Calendar, Contacts, Photos, and Health data values use Data Protection by default, and third-party apps installed on iOS 7 or later receive this protection automatically.

Data Protection is implemented by constructing and managing a hierarchy of keys, and builds on the hardware encryption technologies built into each iOS device. Data Protection is controlled on a per-file basis by assigning each file to a class; accessibility is determined by whether the class keys have been unlocked.

Architecture overview

Every time a file on the data partition is created, Data Protection creates a new 256-bit key (the "per-file" key) and gives it to the hardware AES engine, which uses the key to encrypt the file as it is written to flash memory using AES CBC mode. (On devices with an A8 processor, AES-XTS is used.) The initialization vector (IV) is calculated with the block offset into the file, encrypted with the SHA-1 hash of the per-file key.

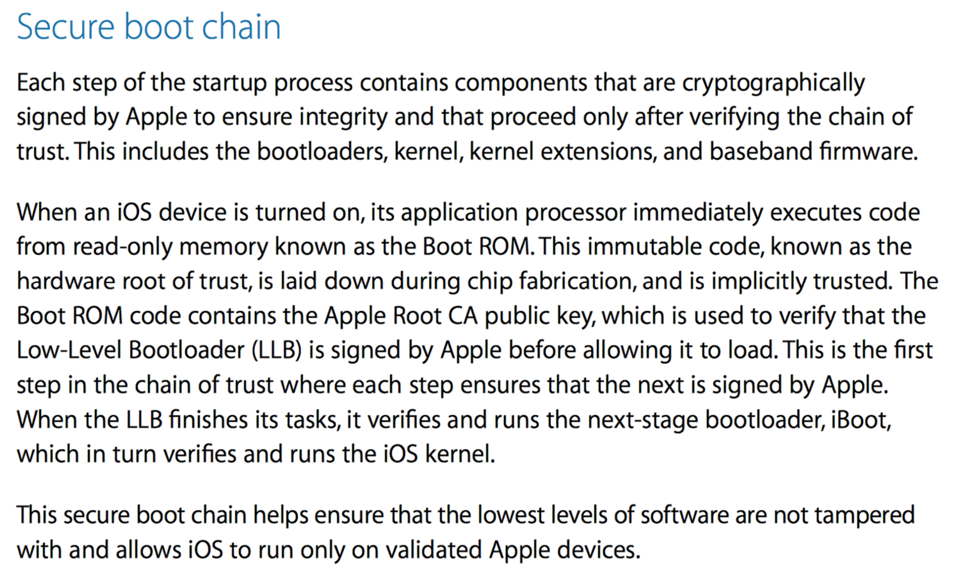
The per-file key is wrapped with one of several class keys, depending on the circumstances under which the file should be accessible. Like all other wrappings, this is performed using NIST AES key wrapping, per RFC 3394. The wrapped per-file key is stored in the file's metadata.

When a file is opened, its metadata is decrypted with the file system key, revealing the wrapped per-file key and a notation on which class protects it. The per-file key is unwrapped with the class key, then supplied to the hardware AES engine, which decrypts the file as it is read from flash memory. All wrapped file key handling occurs in the Secure Enclave; the file key is never directly exposed to the application processor. At boot, the Secure Enclave negotiates an ephemeral key with the AES engine. When the Secure Enclave unwraps a file's keys, they are rewrapped with the ephemeral key and sent back to the application processor.

The metadata of all files in the file system is encrypted with a random key, which is created when iOS is first installed or when the device is wiped by a user. The file system key is stored in Effaceable Storage. Since it's stored on the device, this key is not used to maintain the confidentiality of data; instead, it's designed to be quickly erased on demand (by the user, with the "Erase all content and settings" option, or by a user or administrator issuing a remote wipe command from a mobile device management (MDM) server, Exchange ActiveSync, or iCloud). Erasing the key in this manner renders all files cryptographically inaccessible.

Source: iOS Security Guide, Page 11, https://www.apple.com/business/docs/iOS_Security_Guide.pdf, Last accessed on May 22, 2019

11. The method of claim 10, wherein the program comprises a trusted player and the method further comprises building a manifest for the trusted player, signing the manifest with the asymmetric private key of the predetermined asymmetric key pair, and storing the asymmetric public key of the predetermined asymmetric key pair.	Asana's mobile software application products and services including by way of example, but not limited to the following apps ("mobile applications", "mobile apps" or "Accused Products") that are specifically developed, used, sold, offered for sale, marketed, licensed and distributed by Asana to be downloaded onto Apple iOS mobile or tablet devices. • Asana: organize tasks & work (https://apps.apple.com/us/app/asana-organize-tasks-work/id489969512), Last accessed on Mar 19, 2020 Asana directly infringes and/or continues to knowingly induce Apple to infringe this claim by intentionally developing, making, marketing, advertising, providing, sending, distributing and licensing its mobile applications software, documentation, materials, training or support and aiding, abetting, encouraging, promoting or inviting use thereof. Upon information and belief, the method step of building a manifest for the trusted player, signing the manifest with the asymmetric private key of the predetermined asymmetric key pair, and storing the asymmetric public key of the predetermined asymmetric key pair is performed by Apple and/or its agents – whose acts are attributable to Asana (i) because Asana works together with Apple in a joint enterprise in the building and distribution of its mobile apps, or (ii) because Apple distributes and markets Asana's mobile apps under the direction and control of Asana, or acts as agent, or on behalf of Asana, in the building, marketing and distribution of Asana's mobile apps. Alternatively, to the extent any portion of this method step is performed by Asana, such acts are attributable to Apple, who conditions participation in and the receipt of a benefit, namely, the distribution of Asana's mobile apps through its app store, upon compliance with certain mandatory procedures and guidelines dictated by Apple in the building and upload of Asana's mobile apps, and Asana induces infringement by Apple in the building, marketing and distribution of Asana's mobile apps. As explained in claim 10, iOS implements a secure boot functionality that verifies the operating system, including code for validating digitally signed mobile apps, when the mobile device is powered on. If this verification fails, i.e. if the operating system has been maliciously modified, the operating system, along with the mobile apps installed on the device, does not launch. When Apple installs the bootloader code and the operating system on a device before the device is sold to a user, Apple builds a manifest for the operating system (Apple being the trusted player) and signs the manifest with their asymmetric private key and storing the corresponding asymmetric public key in the bootloader so that the manifest of the operating system can be verified during device restart.

	 Secure boot chain Each step of the startup process contains components that are cryptographically signed by Apple to ensure integrity and that proceed only after verifying the chain of trust. This includes the bootloaders, kernel, kernel extensions, and baseband firmware. When an iOS device is turned on, its application processor immediately executes code from read-only memory known as the Boot ROM. This immutable code, known as the hardware root of trust, is laid down during chip fabrication, and is implicitly trusted. The Boot ROM code contains the Apple Root CA public key, which is used to verify that the Low-Level Bootloader (LLB) is signed by Apple before allowing it to load. This is the first step in the chain of trust where each step ensures that the next is signed by Apple. When the LLB finishes its tasks, it verifies and runs the next-stage bootloader, iBoot, which in turn verifies and runs the iOS kernel. This secure boot chain helps ensure that the lowest levels of software are not tampered with and allows iOS to run only on validated Apple devices. Source: https://www.apple.com/business/docs/iOS_Security_Guide.pdf, Last accessed on May 22, 2019
34. A method of securely distributing data encrypted by a public key of an asymmetric key pair comprising:	Asana's mobile software application products and services including by way of example, but not limited to the following apps ("mobile applications", "mobile apps" or "Accused Products") that are specifically developed, used, sold, offered for sale, marketed, licensed and distributed by Asana to be downloaded onto Apple iOS mobile or tablet devices. • Asana: organize tasks & work (https://apps.apple.com/us/app/asana-organize-tasks-work/id489969512), Last accessed on Mar 19, 2020 Asana directly infringes and/or continues to knowingly induce Apple to infringe this claim by intentionally developing, making, marketing, advertising, providing, sending, distributing and licensing its mobile applications software, documentation, materials, training or support and aiding, abetting, encouraging, promoting or inviting use thereof. To the extent any steps identified herein are performed by Apple, such acts are attributable to Asana (i) because Asana works together with Apple in a joint enterprise in the building and distribution of its mobile apps, or (ii) because Apple distributes and markets Asana's mobile apps under the direction and control of Asana, or acts as agent, or on behalf of Asana, in the building, marketing and distribution of Asana's mobile apps.

	Alternatively, any steps or acts performed by Asana, are attributable to Apple, who conditions participation in and the receipt of a benefit, namely, the distribution of Asana's mobile apps through its app store, upon compliance with certain mandatory procedures and guidelines dictated by Apple in the building and upload of Asana's mobile apps, and Asana induces infringement by Apple in the building, marketing and distribution of Asana's mobile apps. To the extent the preamble is limiting, Asana distributes data encrypted by a public key of an asymmetric key pair. In order to send the mobile app code securely to the Apple servers, data is encrypted with the public key of an asymmetric key pair, as part of the SSL/TLS process. App Store Connect portal (https://appstoreconnect.apple.com, Last accessed on Mar 19, 2020) establishes SSL/TLS communications when uploading Asana's apps, as evidenced by the "https" in the URL. That process necessarily uses the public key of the generated asymmetric key pair to encrypt data that is used to create a symmetric key for secure communications.
building an executable tamper resistant key module identified for a selected program resident on a remote system, the executable tamper resistant key module including a private key of the asymmetric key pair and the encrypted data; and	Relevant to this claim element is the Court's previous construction[30] of "executable tamper resistant key module" / "executable tamper resistant code module" / "tamper resistant key module" to mean "software that is designed to work with other software, that is resistant to observation and modification, and that includes a key for secure communication." Also relevant to this claim element is the Court's previous rejection of limiting "including" to compiling[31]. The method step of "building an executable tamper resistant key module identified for a selected program resident on a remote system, the executable tamper resistant key module including a private key of the asymmetric key pair and the encrypted data" is performed by Asana and/or its agents. To the extent any portion of the method step is performed by Apple and/or its agents, such acts are attributable to Asana (i) because Asana works together with Apple in a joint enterprise in the building and distribution of its mobile apps, or (ii) because Apple distributes and markets Asana's mobile apps under the direction and control of Asana, or acts as agent, or on behalf of Asana, in the building, marketing and distribution of Asana's mobile apps. Building of an "executable tamper resistant code module" (that is, the mobile app) requires the inclusion of at least the following different asymmetric key pairs:

[30] See Memorandum Opinion and Order, Document 104 signed by Judge Rodney Gilstrap on 7/21/2016 in re Plano Encryption Technologies, LLC v. American Bank of Texas (2:15-cv-01273).
[31] Although the exact language construed from the previous claims is not at issue in claim 34, also relevant is the Court's construction of "an asymmetric key pair having a public key and a private key" in claims 1, 9 and 10 to mean "one or more asymmetric key pairs, one of the asymmetric key pairs having the claimed public key and claimed private key, the asymmetric keys of an asymmetric key pair being complementary by performing complementary functions, such as encrypting and decrypting data or creating and verifying signatures." While not necessarily adopting the Court's construction, Honeyman has assumed that the public key and private key in claim 34 must perform complementary functions.

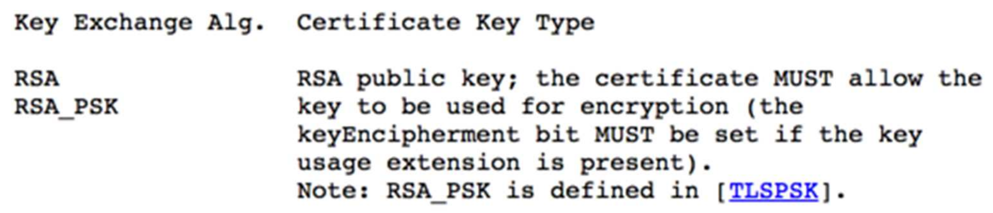
(1) An asymmetric key pair must be included in order to send the mobile app from Asana to Apple securely by SSL/TLS; and
(2) An asymmetric key pair must be included by Asana in order to digitally sign the mobile app with a private asymmetric key and to verify the mobile app has not been changed with the public key for iOS compatible mobile apps.

The asymmetric key pair that is included to digitally sign the mobile app is different from and in addition to the claimed asymmetric key pair used to securely upload the mobile app to the Apple servers for distribution on the Apple App Store.

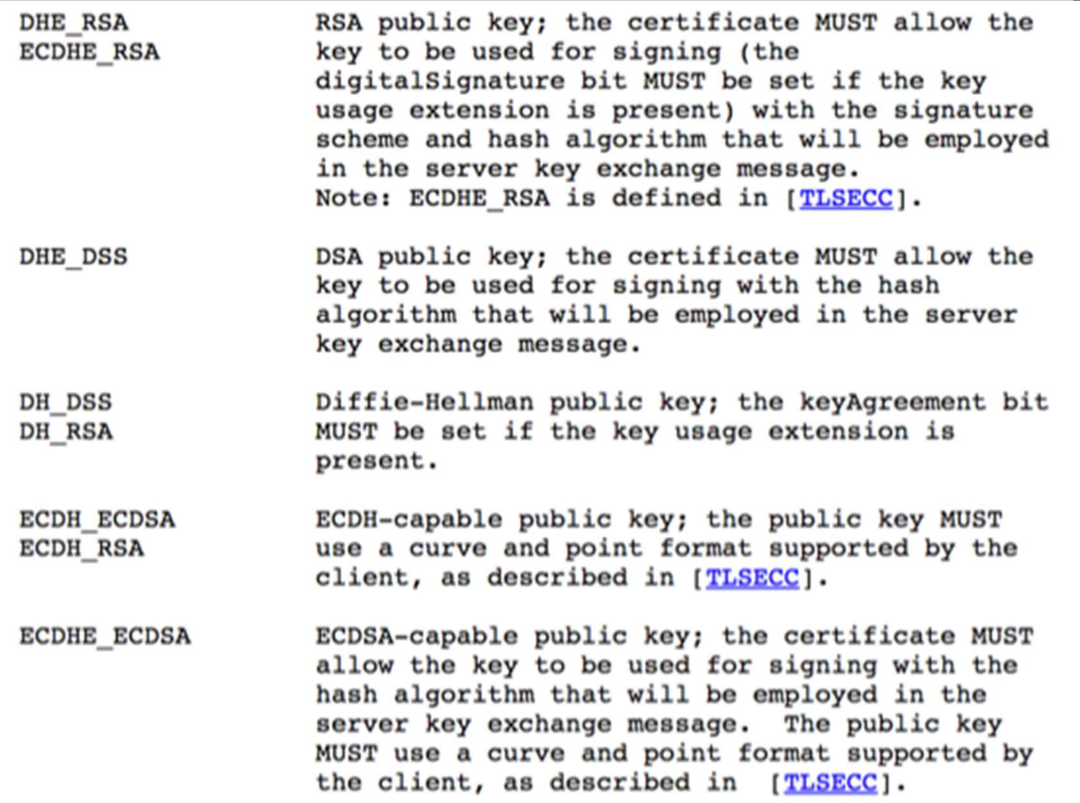
Thus, at least one asymmetric key pair is included having the claimed public key and claimed private key in building the tamper resistant app so that the mobile app code can be sent uploaded to the Apple servers using SSL/TLS protocol. This asymmetric key pair(s) (as with the asymmetric key pair used to digitally sign the app) is complementary as described below by performing complementary functions, such as encrypting and decrypting data and/or creating and verifying signatures. As described in greater detail below, the claimed public and private key are generated and used to securely upload the mobile app onto the Apple servers by SSL/TLS when building the app, where the app includes the generated private key of the claimed asymmetric key pair(s) and the encrypted data.

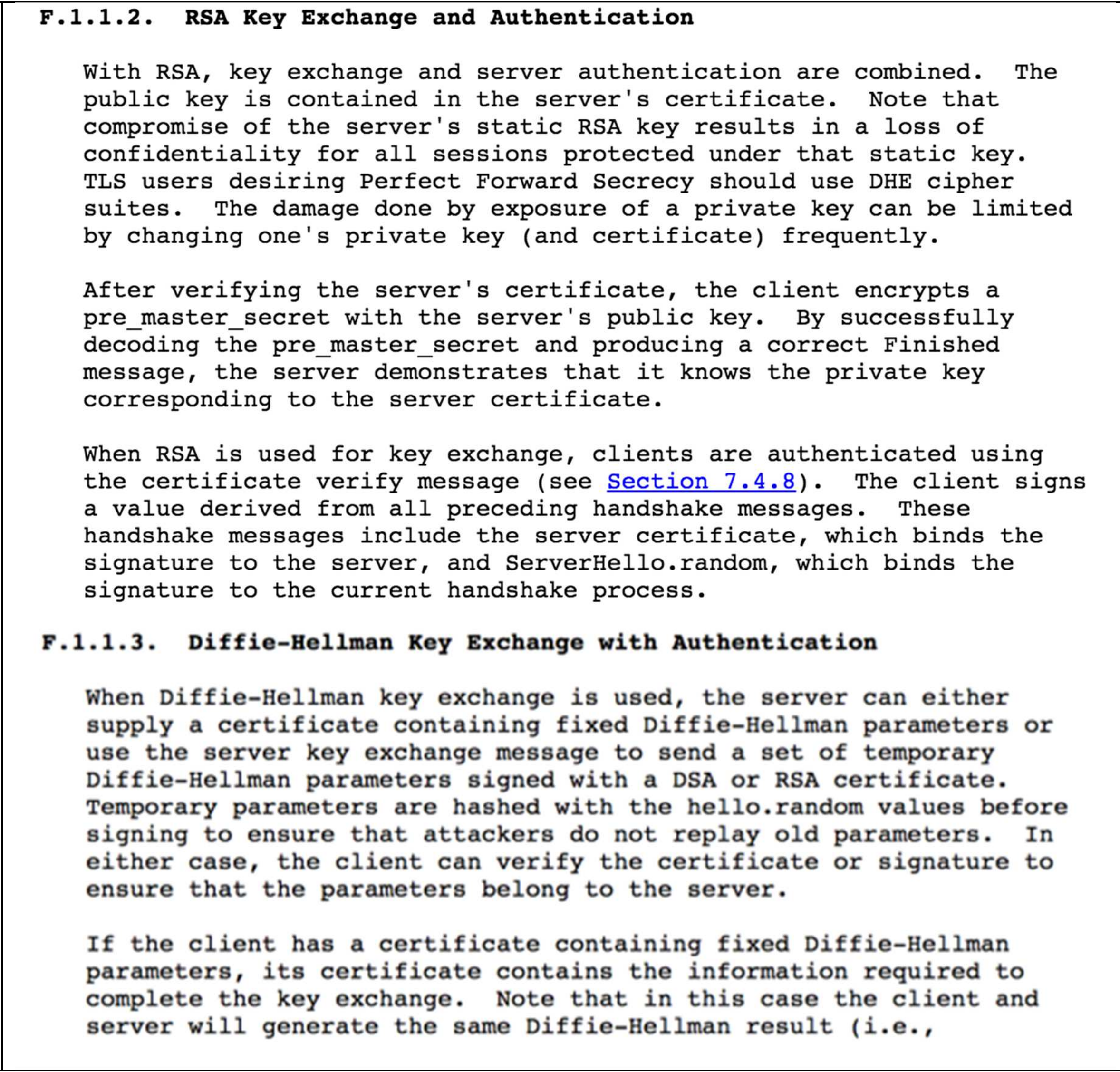
Asana uploads their mobile apps to Apple using a TLS connection – which begins with a TLS handshake. A TLS handshake is a mandatory procedure that allows Asana and Apple to exchange cryptographic parameters, including a cipher suite and arrive at a shared master secret for encrypting communication including upload of Asana’s mobile apps to Apple servers.

A TLS handshake begins with Asana sending a list of cipher suites supported by Asana to Apple. These cipher suites specify at least one or more of the following key exchange algorithms:



```
Key Exchange Alg.  Certificate Key Type

RSA                RSA public key; the certificate MUST allow the
RSA_PSK            key to be used for encryption (the
                   keyEncipherment bit MUST be set if the key
                   usage extension is present).
                   Note: RSA_PSK is defined in [TLSPSK].
```

	 DHE_RSA ECDHE_RSA — RSA public key; the certificate MUST allow the key to be used for signing (the digitalSignature bit MUST be set if the key usage extension is present) with the signature scheme and hash algorithm that will be employed in the server key exchange message. Note: ECDHE_RSA is defined in [TLSECC]. DHE_DSS — DSA public key; the certificate MUST allow the key to be used for signing with the hash algorithm that will be employed in the server key exchange message. DH_DSS DH_RSA — Diffie-Hellman public key; the keyAgreement bit MUST be set if the key usage extension is present. ECDH_ECDSA ECDH_RSA — ECDH-capable public key; the public key MUST use a curve and point format supported by the client, as described in [TLSECC]. ECDHE_ECDSA — ECDSA-capable public key; the certificate MUST allow the key to be used for signing with the hash algorithm that will be employed in the server key exchange message. The public key MUST use a curve and point format supported by the client, as described in [TLSECC]. *Source: https://tools.ietf.org/html/rfc5246 at 48-49*, Last accessed on Mar 19, 2020



F.1.1.2. RSA Key Exchange and Authentication

With RSA, key exchange and server authentication are combined. The public key is contained in the server's certificate. Note that compromise of the server's static RSA key results in a loss of confidentiality for all sessions protected under that static key. TLS users desiring Perfect Forward Secrecy should use DHE cipher suites. The damage done by exposure of a private key can be limited by changing one's private key (and certificate) frequently.

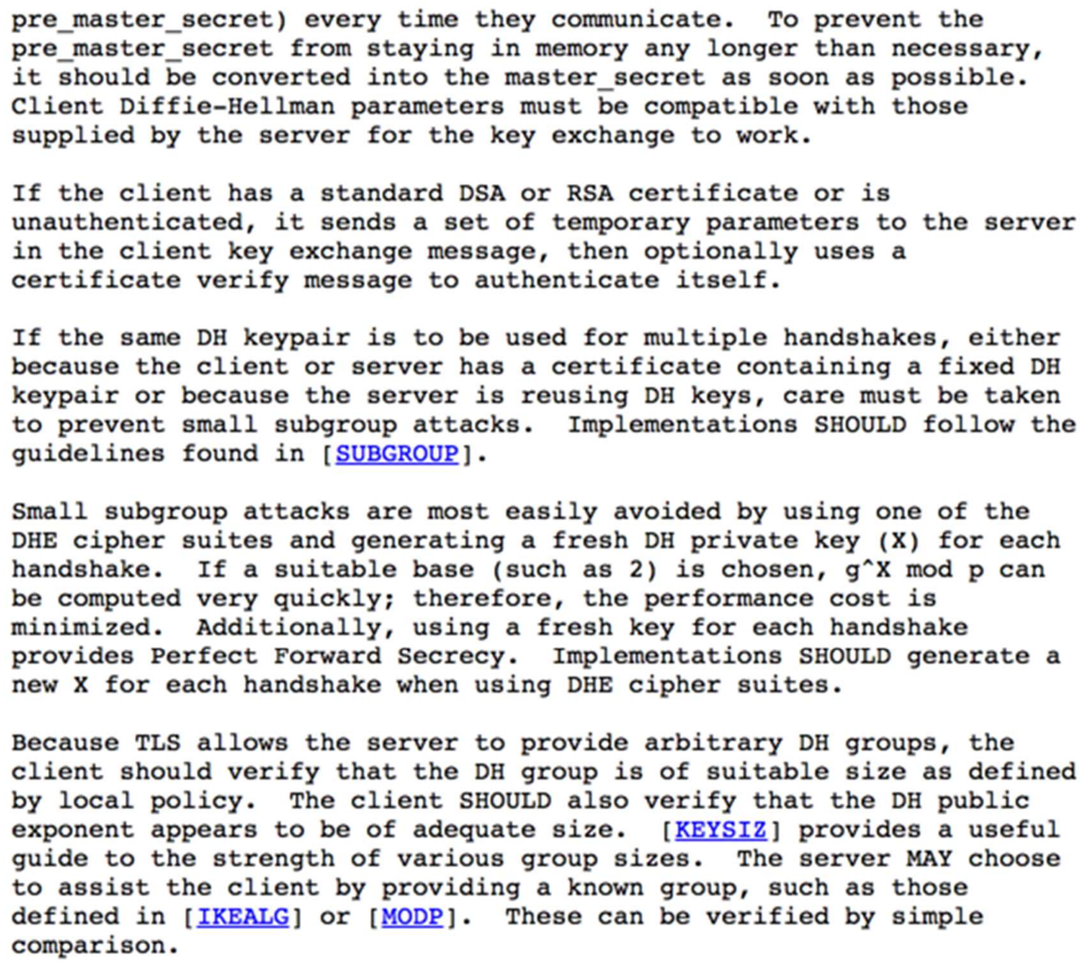
After verifying the server's certificate, the client encrypts a pre_master_secret with the server's public key. By successfully decoding the pre_master_secret and producing a correct Finished message, the server demonstrates that it knows the private key corresponding to the server certificate.

When RSA is used for key exchange, clients are authenticated using the certificate verify message (see Section 7.4.8). The client signs a value derived from all preceding handshake messages. These handshake messages include the server certificate, which binds the signature to the server, and ServerHello.random, which binds the signature to the current handshake process.

F.1.1.3. Diffie-Hellman Key Exchange with Authentication

When Diffie-Hellman key exchange is used, the server can either supply a certificate containing fixed Diffie-Hellman parameters or use the server key exchange message to send a set of temporary Diffie-Hellman parameters signed with a DSA or RSA certificate. Temporary parameters are hashed with the hello.random values before signing to ensure that attackers do not replay old parameters. In either case, the client can verify the certificate or signature to ensure that the parameters belong to the server.

If the client has a certificate containing fixed Diffie-Hellman parameters, its certificate contains the information required to complete the key exchange. Note that in this case the client and server will generate the same Diffie-Hellman result (i.e.,



pre_master_secret) every time they communicate. To prevent the pre_master_secret from staying in memory any longer than necessary, it should be converted into the master_secret as soon as possible. Client Diffie-Hellman parameters must be compatible with those supplied by the server for the key exchange to work.

If the client has a standard DSA or RSA certificate or is unauthenticated, it sends a set of temporary parameters to the server in the client key exchange message, then optionally uses a certificate verify message to authenticate itself.

If the same DH keypair is to be used for multiple handshakes, either because the client or server has a certificate containing a fixed DH keypair or because the server is reusing DH keys, care must be taken to prevent small subgroup attacks. Implementations SHOULD follow the guidelines found in [SUBGROUP].

Small subgroup attacks are most easily avoided by using one of the DHE cipher suites and generating a fresh DH private key (X) for each handshake. If a suitable base (such as 2) is chosen, g^X mod p can be computed very quickly; therefore, the performance cost is minimized. Additionally, using a fresh key for each handshake provides Perfect Forward Secrecy. Implementations SHOULD generate a new X for each handshake when using DHE cipher suites.

Because TLS allows the server to provide arbitrary DH groups, the client should verify that the DH group is of suitable size as defined by local policy. The client SHOULD also verify that the DH public exponent appears to be of adequate size. [KEYSIZ] provides a useful guide to the strength of various group sizes. The server MAY choose to assist the client by providing a known group, such as those defined in [IKEALG] or [MODP]. These can be verified by simple comparison.

Source: The Transport Layer Security (TLS) Protocol Version 1.2, IETF RFC 5246, https://tools.ietf.org/html/rfc5246, Last accessed on Mar 19, 2020

For each of the key exchange algorithms, Asana and/or Apple build an executable tamper resistant key module that includes the generated private key and the encrypted data.

	For **RSA and RSA _PSK**, the executable tamper resistant key module includes encrypted data (i.e. the encrypted premaster secret) as it would be impossible for Asana to send its mobile app to Apple using SSL/TLS without encrypting a premaster secret and sending it to Apple. The executable tamper resistant key module also includes: 1. Apple's RSA private key corresponding to Apple's RSA public key used by Asana to encrypt the premaster secret. 2. Asana's private key used to code-sign the mobile app. For **DHE_RSA, ECDHE_RSA, DH_RSA and ECDH_RSA**, the executable tamper resistant key module includes encrypted data (i.e. all communication with Apple subsequent to the TLS handshake, including at least the executable compiled code related to its mobile apps and information such as name, category, screenshots and description related to Asana's mobile apps). The executable tamper resistant key module also includes: 1. Apple's Diffie-Hellman private key corresponding to Apple's Diffie-Hellman public key used by Asana to compute the shared master secret. 2. Asana's Diffie-Hellman private key used to compute the shared master secret. 3. Apple's RSA private key used to sign the message containing Apple's Diffie-Hellman public key. 4. Asana's private key used to code-sign the mobile app. For **DHE_DSS and DH_DSS**, the executable tamper resistant key module includes encrypted data (i.e. all communication with Apple subsequent to the TLS handshake, including at least the executable compiled code related to its mobile apps and information such as name, category, screenshots and description related to Asana's mobile apps). The executable tamper resistant key module also includes: 1. Apple's Diffie-Hellman private key corresponding to Apple's Diffie-Hellman public key used by Asana to compute the shared master secret. 2. Asana's Diffie-Hellman private key used to compute the shared master secret. 3. Apple's DSA private key used to sign the message containing Apple's Diffie-Hellman public key. 4. Asana's private key used to code-sign the mobile app. For **ECDH_ECDSA and ECDHE_ECDSA**, the executable tamper resistant key module includes encrypted data (i.e. all communication with Apple subsequent to the TLS handshake, including at least the executable compiled code related to its mobile apps and information such as name, category, screenshots and description related to Asana's mobile apps). The executable tamper resistant key module also includes: 1. Apple's Diffie-Hellman private key corresponding to Apple's Diffie-Hellman public key used by Asana to compute the shared master secret. 2. Asana's Diffie-Hellman private key used to compute the shared master secret. 3. Apple's ECDSA private key used to sign the message containing Apple's Diffie-Hellman public key. 4. Asana's private key used to code-sign the mobile app. Asana builds an executable tamper resistant key module identified for a selected program resident on a remote system. Specifically, Asana builds a mobile app, which is an executable tamper resistant key module, as explained in more detail below. This mobile app is identified for a selected program resident on a remote system, namely the iOS operating system on a remote mobile device.

The mobile app comprises an executable tamper resistant key module that is identified for either the iOS program and includes the claimed private key described above and the encrypted data encrypted with the claimed public key also described above in building the mobile app so that it can be made available for download from Apple servers onto devices compatible with iOS for use by customers of Asana. An asymmetric key pair is used not only to upload the binary code files for the mobile app, but an entire application package, including all of the metadata for the app, such as title, screenshots, and other resources or information such as application type, category, price, *etc*. which are included during the upload process so that the mobile app can be identified by potential users for download[32].

Asana's mobile app is each an executable tamper resistant key module because it is designed to work with other software, namely the iOS operating system as well as other applications or programs installed on a user's mobile device; because the mobile app is resistant to observation and modification, as explained below; and because in building Asana mobile apps on the Apple platform, Asana's apps include at least the claimed generated private key and the encrypted data including by way of example, the pre-master secret encrypted with the claimed generated public key when the mobile app is securely uploaded onto the Apple servers as described above.

The tamper resistant key module includes several keys "used for secure communications" per the Court's previous construction including at least the following:
For **RSA and RSA _PSK**:
1. Apple's RSA private key corresponding to Apple's RSA public key used by Asana to encrypt the premaster secret.
2. Asana's private key used to code-sign the mobile app.
3. Symmetric key used for uploading the mobile app to Apple subsequent to the TLS handshake.
4. Asymmetric keys and symmetric keys used for TLS communications during operation of the app.

For **DHE_RSA, ECDHE_RSA, DH_RSA and ECDH_RSA**:
1. Apple's Diffie-Hellman private key corresponding to Apple's Diffie-Hellman public key used by Asana to compute the shared master secret.
2. Asana's Diffie-Hellman private key used to compute the shared master secret.
3. Apple's RSA private key used to sign the message containing Apple's Diffie-Hellman public key.
4. Asana's private key used to code-sign the mobile app.
5. Shared master secret key used for uploading the mobile app to Apple subsequent to the TLS handshake.
6. Asymmetric keys and symmetric keys used for TLS communications during operation of the app.

For **DHE_DSS and DH_DSS**:
1. Apple's Diffie-Hellman private key corresponding to Apple's Diffie-Hellman public key used by Asana to compute the shared master secret.
2. Asana's Diffie-Hellman private key used to compute the shared master secret.

[32] *See, e.g.*, https://help.apple.com/xcode/mac/current/#/dev067853c94, Last accessed on Mar 19, 2020

3. Apple's DSA private key used to sign the message containing Apple's Diffie-Hellman public key.
4. Asana's private key used to code-sign the mobile app.
5. Shared master secret key used for uploading the mobile app to Apple subsequent to the TLS handshake.
6. Asymmetric keys and symmetric keys used for TLS communications during operation of the app.

For **ECDH_ECDSA and ECDHE_ECDSA**:

1. Apple's Diffie-Hellman private key corresponding to Apple's Diffie-Hellman public key used by Asana to compute the shared master secret.
2. Asana's Diffie-Hellman private key used to compute the shared master secret.
3. Apple's ECDSA private key used to sign the message containing Apple's Diffie-Hellman public key.
4. Asana's private key used to code-sign the mobile app.
5. Shared master secret key used for uploading the mobile app to Apple subsequent to the TLS handshake.
6. Asymmetric keys and symmetric keys used for TLS communications during operation of the app.

Asana's mobile apps are tamper resistant, resistant to observation and modification, as follows:

1. **Resistant to Observation Because App Source Code Is Compiled Before Upload**

Asana mobile apps are resistant to observation, at least in part, since Asana compiles its mobile app source code before submitting the app to Apple – and uploads the binary output of the compilation process rather than the source code itself[33].

See, e.g., App Store Connect Help**,** **"**Submit your app for review**"** stating **"**You submit your app to App Review to start the review process for making your app available on the App Store. However, before you can submit an app to App Review, you must provide the required metadata and choose the build for the version.
Before you submit an app to App Review, you choose which build (from all the builds you uploaded for a version) that you want to submit. You can associate only one build with an App Store version. However, you can change the build as often as you want until you submit the version to App Review."

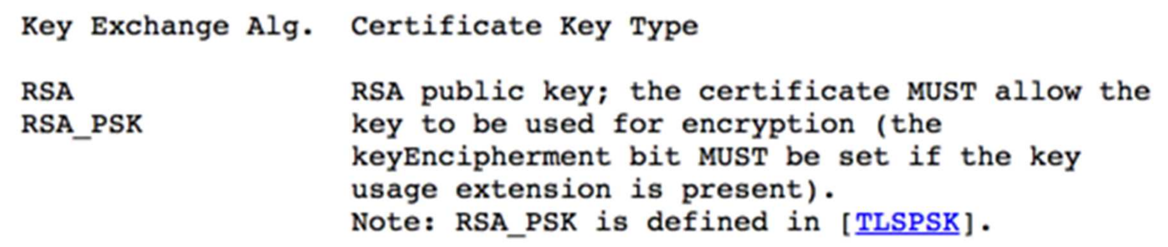
Source: https://help.apple.com/app-store-connect/#/dev301cb2b3e, Last accessed on Mar 19, 2020

[33] *See, e.g.,* https://help.apple.com/app-store-connect/#/dev301cb2b3e, Last accessed on Mar 19, 2020

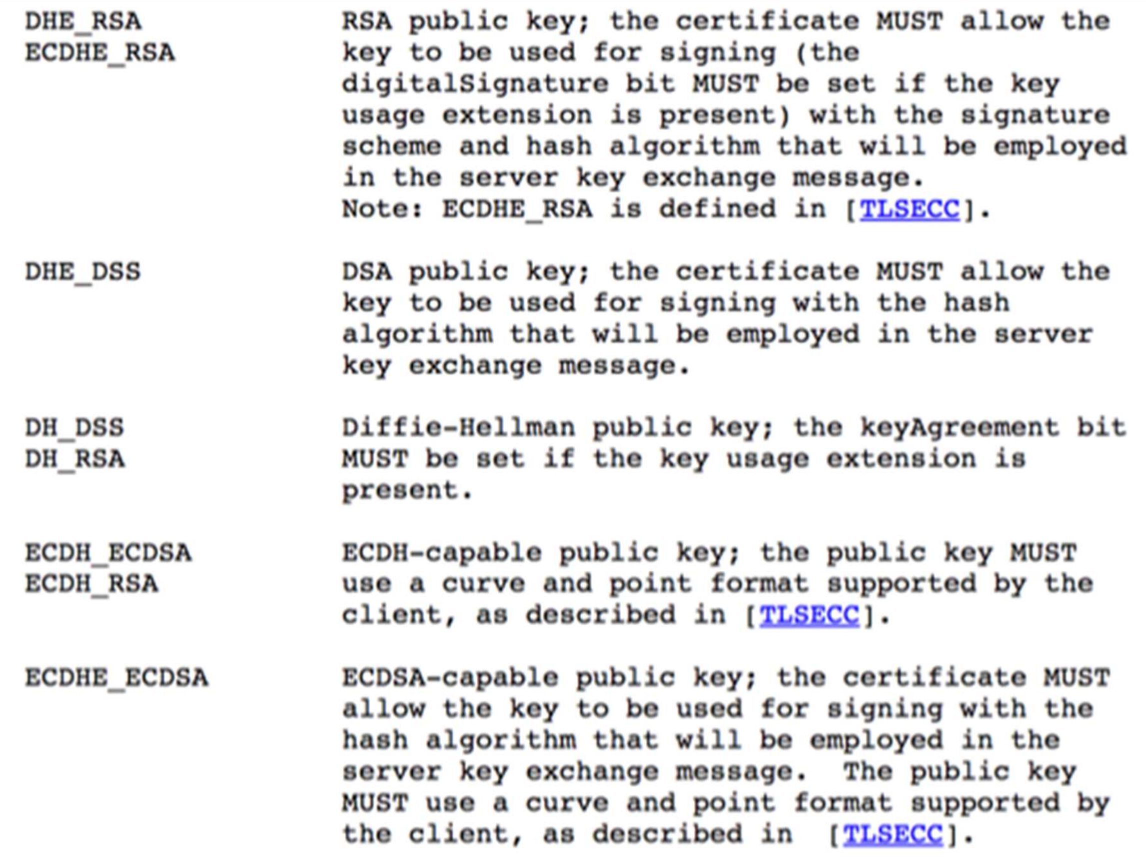
2. Resistant to Observation Because Upload To Apple Is Over SSL/TLS

Asana’s mobile apps are made further resistant to observation, at least in part, because the mobile app is securely sent by SSL/TLS to Apple as part of the building process. App Store Connect portal (https://appstoreconnect.apple.com, Last accessed on Mar 19, 2020) establishes SSL/TLS communications when uploading Asana’s apps, as evidenced by the “https” in its URL. Sending the mobile app code by SSL/TLS is necessary to keep the code from being observed in transit from the code developer to Apple.

The secure upload process starts with a TLS handshake procedure which uses asymmetric key encryption and necessitates generation of at least one asymmetric key pair. Specifically, Asana negotiates with Apple the cipher suite and the key exchange algorithm that will be used for the handshake.

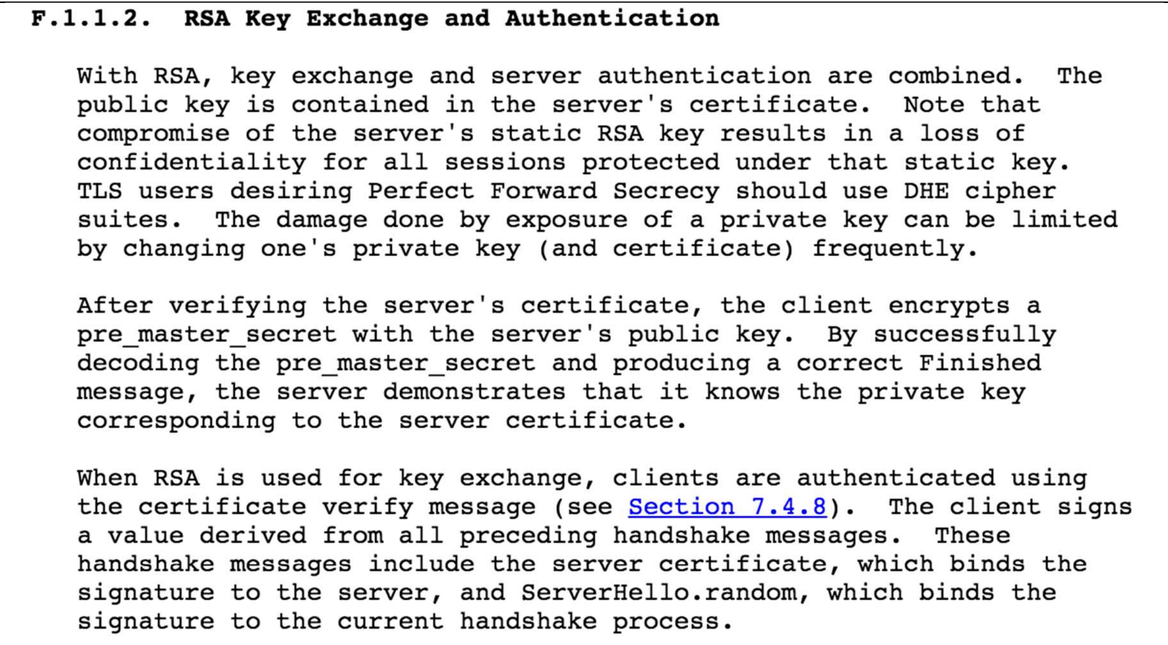


```
Key Exchange Alg.  Certificate Key Type

RSA                RSA public key; the certificate MUST allow the
RSA_PSK            key to be used for encryption (the
                   keyEncipherment bit MUST be set if the key
                   usage extension is present).
                   Note: RSA_PSK is defined in [TLSPSK].
```



DHE_RSA ECDHE_RSA	RSA public key; the certificate MUST allow the key to be used for signing (the digitalSignature bit MUST be set if the key usage extension is present) with the signature scheme and hash algorithm that will be employed in the server key exchange message. Note: ECDHE_RSA is defined in [TLSECC].
DHE_DSS	DSA public key; the certificate MUST allow the key to be used for signing with the hash algorithm that will be employed in the server key exchange message.
DH_DSS DH_RSA	Diffie-Hellman public key; the keyAgreement bit MUST be set if the key usage extension is present.
ECDH_ECDSA ECDH_RSA	ECDH-capable public key; the public key MUST use a curve and point format supported by the client, as described in [TLSECC].
ECDHE_ECDSA	ECDSA-capable public key; the certificate MUST allow the key to be used for signing with the hash algorithm that will be employed in the server key exchange message. The public key MUST use a curve and point format supported by the client, as described in [TLSECC].

Source: https://tools.ietf.org/html/rfc5246 at 48-49, Last accessed on Mar 19, 2020

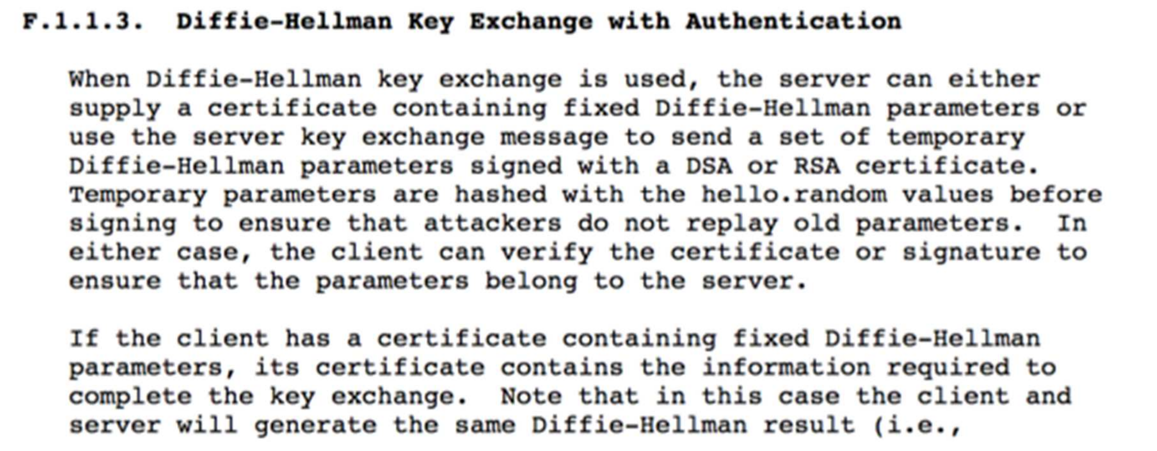


F.1.1.2. RSA Key Exchange and Authentication

With RSA, key exchange and server authentication are combined. The public key is contained in the server's certificate. Note that compromise of the server's static RSA key results in a loss of confidentiality for all sessions protected under that static key. TLS users desiring Perfect Forward Secrecy should use DHE cipher suites. The damage done by exposure of a private key can be limited by changing one's private key (and certificate) frequently.

After verifying the server's certificate, the client encrypts a pre_master_secret with the server's public key. By successfully decoding the pre_master_secret and producing a correct Finished message, the server demonstrates that it knows the private key corresponding to the server certificate.

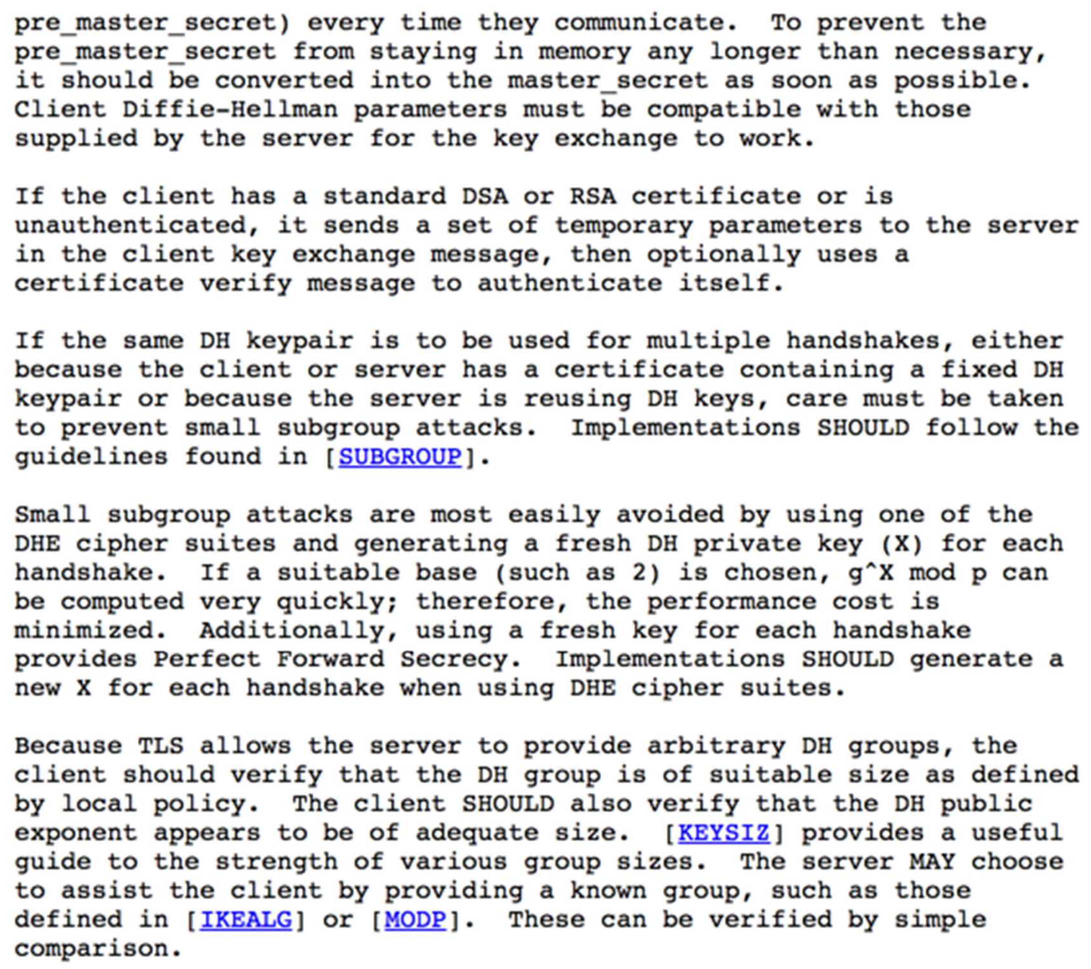
When RSA is used for key exchange, clients are authenticated using the certificate verify message (see Section 7.4.8). The client signs a value derived from all preceding handshake messages. These handshake messages include the server certificate, which binds the signature to the server, and ServerHello.random, which binds the signature to the current handshake process.



F.1.1.3. Diffie-Hellman Key Exchange with Authentication

When Diffie-Hellman key exchange is used, the server can either supply a certificate containing fixed Diffie-Hellman parameters or use the server key exchange message to send a set of temporary Diffie-Hellman parameters signed with a DSA or RSA certificate. Temporary parameters are hashed with the hello.random values before signing to ensure that attackers do not replay old parameters. In either case, the client can verify the certificate or signature to ensure that the parameters belong to the server.

If the client has a certificate containing fixed Diffie-Hellman parameters, its certificate contains the information required to complete the key exchange. Note that in this case the client and server will generate the same Diffie-Hellman result (i.e.,



pre_master_secret) every time they communicate. To prevent the pre_master_secret from staying in memory any longer than necessary, it should be converted into the master_secret as soon as possible. Client Diffie-Hellman parameters must be compatible with those supplied by the server for the key exchange to work.

If the client has a standard DSA or RSA certificate or is unauthenticated, it sends a set of temporary parameters to the server in the client key exchange message, then optionally uses a certificate verify message to authenticate itself.

If the same DH keypair is to be used for multiple handshakes, either because the client or server has a certificate containing a fixed DH keypair or because the server is reusing DH keys, care must be taken to prevent small subgroup attacks. Implementations SHOULD follow the guidelines found in [SUBGROUP].

Small subgroup attacks are most easily avoided by using one of the DHE cipher suites and generating a fresh DH private key (X) for each handshake. If a suitable base (such as 2) is chosen, g^X mod p can be computed very quickly; therefore, the performance cost is minimized. Additionally, using a fresh key for each handshake provides Perfect Forward Secrecy. Implementations SHOULD generate a new X for each handshake when using DHE cipher suites.

Because TLS allows the server to provide arbitrary DH groups, the client should verify that the DH group is of suitable size as defined by local policy. The client SHOULD also verify that the DH public exponent appears to be of adequate size. [KEYSIZ] provides a useful guide to the strength of various group sizes. The server MAY choose to assist the client by providing a known group, such as those defined in [IKEALG] or [MODP]. These can be verified by simple comparison.

Source: The Transport Layer Security (TLS) Protocol Version 1.2, IETF RFC 5246, https://tools.ietf.org/html/rfc5246, Last accessed on Mar 19, 2020

For each of the above key exchange algorithms, Apple and/or Asana encrypts all communication, including upload of the mobile app, using a master secret, rendering the communication resistant to observation during transit from Asana to Apple.

For **RSA and RSA _PSK**, Asana encrypts a random premaster secret with Apple's RSA public key and sends the encrypted premaster secret to Apple. Apple decrypts the premaster secret with its matched RSA private key. Asana and Apple both use the premaster secret to compute a master secret which is then used by both Asana and Apple to encrypt all subsequent communications between Asana and Apple.

For the other key exchange algorithms, namely Diffie-Hellman based algorithms such as **DHE_RSA, ECDHE_RSA, DH_RSA, DHE_DSS, DH_DSS, ECDH_RSA**, **ECDH_ECDSA and ECDHE_ECDSA,** Asana sends its Diffie-Hellman public key[34] to Apple while Apple sends its Diffie-Hellman public key to Asana. Asana then uses Apple's Diffie-Hellman public key combined with Asana's own Diffie-Hellman private key to compute a premaster secret. Apple in turn uses Asana's Diffie-Hellman public key combined with Apple's own Diffie-Hellman private key to compute the same premaster secret. Asana and Apple both use the premaster secret to compute a master secret which is then used by both Asana and Apple to encrypt all subsequent communications between Asana and Apple.

3. Resistant to Modification Because App Binary Is Code Signed

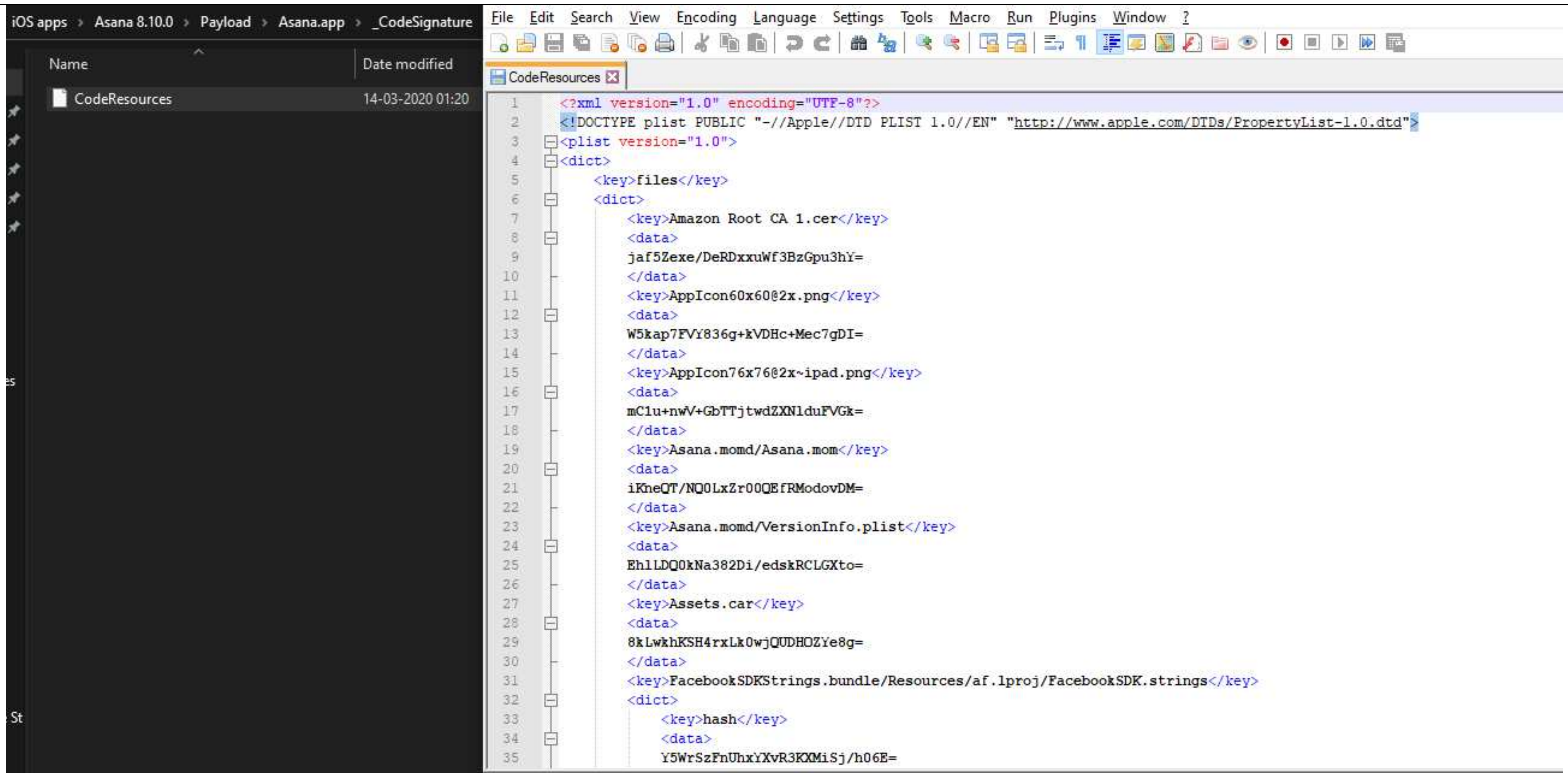
The mobile app code is made resistant to modification, at least in part, because the app binary is code signed. Apple dictates that each developer must sign the mobile app submission with his/her asymmetric developer key that certifies that the app has not been modified by a third party impersonator[35].

> *Xcode code signs your app during the build and archive process. If needed, Xcode requests a certificate and adds a signing certificate, the certificate with its public-private key pair, to your keychain. The certificate with the public key is added to your developer account.*
>
> Source: https://help.apple.com/xcode/mac/current/#/dev3a05256b8, Last accessed on Mar 19, 2020
>
> *A signing signing certificate includes the certificate with its public-private key pair issued by Apple, and is stored in your keychain. Because the private key is stored locally, protect it as you would an account password. An intermediate certificate is also required to be in your keychain to ensure that your certificate is issued by a certificate authority such as Apple.*
> *Your signing certificate is added to your keychain and the corresponding certificate is added to your developer account.*
>
> *Source:* https://help.apple.com/xcode/mac/current/#/dev1c7c2c67d, Last accessed on Mar 19, 2020

[34] For Diffie-Hellman (DH) based algorithms such as DH_RSA, DHE_RSA, ECDH_RSA, ECDHE_RSA, DH_DSS, DHE_DSS, ECDH_ECDSA and ECDHE_ECDSA, Apple calculates a hash of the message containing their Diffie-Hellman public key and encrypts the hash with their RSA/DSA/ECDSA private key (i.e. signing the message). Apple then sends that RSA/DSA/ECDSA public key to Asana in a digital certificate so that Asana can authenticate the Apple server by decrypting the hash using Apple's public key and matching the decryption result to a hash of the received message as calculated by Asana itself. If the two values match, Asana knows that the message originated from Apple and not from a malicious third party.

[35] *See, e.g.,* https://help.apple.com/xcode/mac/current/#/dev3a05256b8, Last accessed on Mar 19, 2020

	About Signing Identities and Certificates *Code signing (or signing) an app allows the system to identify who signed the app and to verify that the app has not been modified since it was signed.* *Signing is a requirement for uploading your app to App Store Connect and distributing it through TestFlight or the App Store. The operating system verifies the signature of apps downloaded from the App Store to ensure that apps with invalid signatures don't run. An app's executable code is protected by its signature because the signature becomes invalid if any of the executable code in the app bundle changes. A valid signature lets users trust that the app was signed by an Apple source and hasn't been modified since it was signed.* *Xcode uses your signing certificate to sign your app during the build process. The signing certificate consists of a public-private key pair and a certificate. The private key is used by cryptographic functions to generate the signature. The certificate is issued by Apple; it contains the public key and identifies you as the owner of the key pair. In order to sign apps, you must have both parts of your signing certificate, and an Apple certificate authority in your keychain.* *An app's signature can be removed, and the app can be re-signed using another signing certificate. For example, Apple re-signs all apps sold on the App Store. Also, a fully-tested development build of your app can be re-signed for submission to the App Store. Thus the signature is best understood not as proof of the app's origin but as a verifiable mark placed by the signer.* Source: https://help.apple.com/xcode/mac/current/#/devfbe995ebf, Last accessed on Mar 19, 2020 Asana complies with Apple's instructions on code signing as shown by Asana's mobile app contents. Asana's mobile apps contain files such as the file _CodeSignature/CodeResources in Asana's iOS apps which are generated during the code signing process as per instructions from Apple.

iOS apps › Asana 8.10.0 › Payload › Asana.app › _CodeSignature
Name
Date modified
CodeResources
14-03-2020 01:20
File Edit Search View Encoding Language Settings Tools Macro Run Plugins Window ?
CodeResources

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN" "http://www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
<dict>
    <key>files</key>
    <dict>
        <key>Amazon Root CA 1.cer</key>
        <data>
        jaf5Zexe/DeRDxxuWf3BzGpu3hY=
        </data>
        <key>AppIcon60x60@2x.png</key>
        <data>
        W5kap7FVY836g+kVDHc+Mec7gDI=
        </data>
        <key>AppIcon76x76@2x~ipad.png</key>
        <data>
        mC1u+nwV+GbTTjtwdZXNlduFVGk=
        </data>
        <key>Asana.momd/Asana.mom</key>
        <data>
        iKneQT/NQ0LxZr00QEfRModovDM=
        </data>
        <key>Asana.momd/VersionInfo.plist</key>
        <data>
        Eh1LDQ0kNa382Di/edskRCLGXto=
        </data>
        <key>Assets.car</key>
        <data>
        8kLwkhKSH4rxLk0wjQUDHOZYe8g=
        </data>
        <key>FacebookSDKStrings.bundle/Resources/af.lproj/FacebookSDK.strings</key>
        <dict>
            <key>hash</key>
            <data>
            Y5WrSzFnUhxYXvR3KXMiSj/h06E=
```

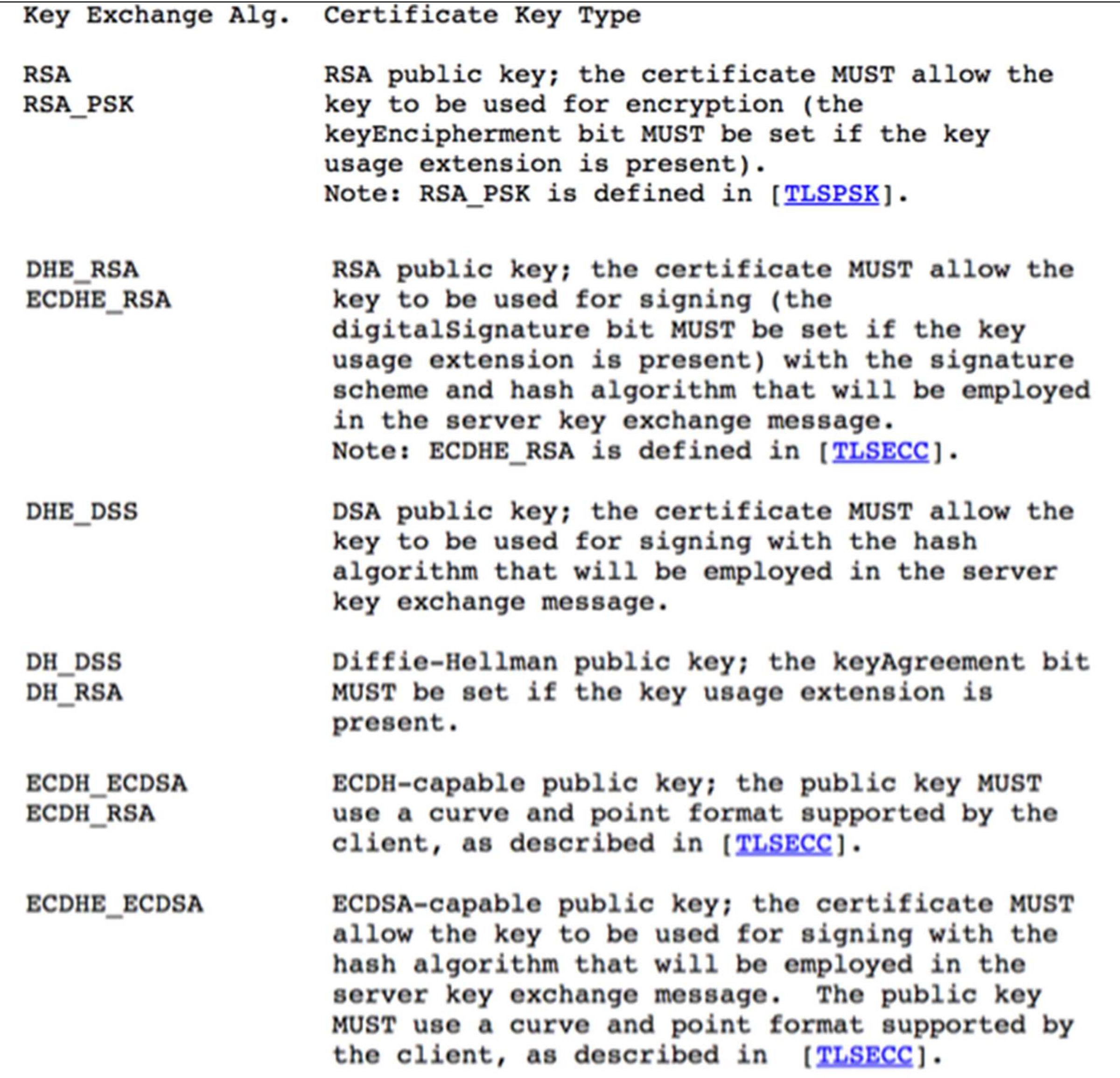
Source: Contents of Asana: organize tasks & work, https://apps.apple.com/us/app/asana-organize-tasks-work/id489969512, as an example of Asana app, Last accessed on Mar 19, 2020

Asymmetrical key cryptography and hashing algorithms are used to create the unique digital signature for iOS mobile apps. The digital signature is used to sign the resources in an application package, including the compiled code. The private key of an asymmetric key pair that is generated for the digital code signing is used to code sign the app. This private key is included in the mobile app although the private key is not the claimed private key of the claimed generated asymmetric key pair because it does not match the claimed public key used to encrypt data, based on the court's construction.

	Hashes are created for every resource in the application package with the help of a hash algorithm. The signature manifest also has its own hash to prevent unauthorized changes. The hashes are encrypted with a private key. After the encryption is complete, the digital signature for the app is created. By signing the app binary with a digital signature, Asana's mobile apps are tamper resistant enabling Apple and the iOS mobile devices to verify that the application is being distributed by trusted source (*i.e.* Asana) and that the application has not been modified by a third party, which can be verified by the corresponding public key generated as part of the pair. Thus the app binary is made resistant to modification by digital signing. Accordingly Asana's iOS mobile apps establish SSL/TLS communications with Asana's servers, which involve a SSL/TLS handshake procedure involving asymmetric key encryption. SSL/TLS ensures secure communication and renders the mobile app data further resistant to observation.
sending the executable tamper resistant key module to the remote system.	Upon information and belief, the method step of sending the executable tamper resistant key module to the remote system is performed by Apple and/or its agents – whose acts are attributable to Asana (i) because Asana works together with Apple in a joint enterprise in the building and distribution of its mobile apps, or (ii) because Apple distributes and markets Asana's mobile apps under the direction and control of Asana, or acts as agent, or on behalf of Asana, in the building, marketing and distribution of Asana's mobile apps. Alternatively, to the extent any portion of this method step is performed by Asana, such acts are attributable to Apple, who conditions participation in and the receipt of a benefit, namely, the distribution of Asana's mobile apps through its app store, upon compliance with certain mandatory procedures and guidelines dictated by Apple in the building and upload of Asana's mobile apps, and Asana induces infringement by Apple in the building, marketing and distribution of Asana's mobile apps. Asana's mobile apps are sent or downloaded from Apple servers and are executed on iOS remote devices such as mobile phones and tablets. When a user accesses Apple App Store – and requests to download Asana app, Apple sends the executable tamper resistant key module to the remote device(s). Further, the step of "sending" Asana mobile apps to the remote system occurs via TLS/SSL communications In particular, Asana mobile apps sent to users' remote devices are tamper resistant, resistant to observation and modification as follows: **1. Resistant to Observation Because App is Downloaded in Compiled Form**

	Asana mobile apps are resistant to observation, at least in part, since Asana compiles its mobile app source code before submitting the app to Apple – and uploads the binary output of the compilation process rather than the source code itself – and hence a user can only download the compiled source code from Apple rather than the source code itself[36]. See, e.g., App Store Connect Help**,** **“**Submit your app for review**”** stating **“**You submit your app to App Review to start the review process for making your app available on the App Store. However, before you can submit an app to App Review, you must provide the required metadata and choose the build for the version. Before you submit an app to App Review, you choose which build (from all the builds you uploaded for a version) that you want to submit. You can associate only one build with an App Store version. However, you can change the build as often as you want until you submit the version to App Review.” Source: https://help.apple.com/app-store-connect/#/dev301cb2b3e, Last accessed on Mar 19, 2020 **2. Resistant to Observation Because Download from Apple Is Over SSL/TLS** Asana’s mobile apps are made further resistant to observation, at least in part, because the mobile app is securely sent or downloaded by SSL/TLS from Apple servers. Asana app users establish SSL/TLS communications with Apple App Store (for example using the URL https://apps.apple.com/us/app/asana-organize-tasks-work/id489969512 for Asana: organize tasks & work, Last accessed on Mar 19, 2020) when downloading Asana’s iOS apps, as evidenced by the “https” in the URL. Sending the mobile app code by SSL/TLS is necessary to keep the code from being observed in transit from Apple to the user’s remote system. The secure download process starts with a TLS handshake procedure which uses asymmetric key encryption and necessitates generation of at least one asymmetric key pair. Specifically, user’s remote device negotiates with Apple the cipher suite and the key exchange algorithm that will be used for the handshake.

[36] *See, e.g.,* https://help.apple.com/app-store-connect/#/dev301cb2b3e, Last accessed on Mar 19, 2020

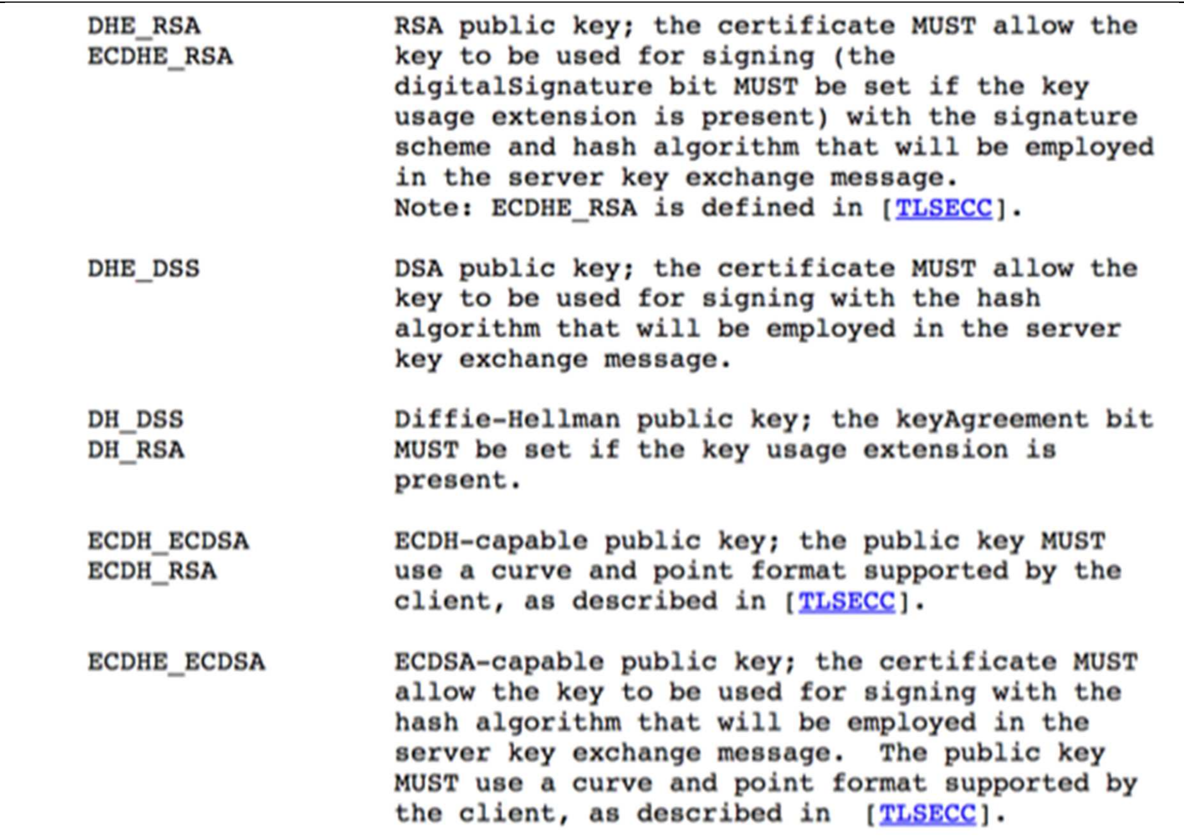


Key Exchange Alg.	Certificate Key Type
RSA RSA_PSK	RSA public key; the certificate MUST allow the key to be used for encryption (the keyEncipherment bit MUST be set if the key usage extension is present). Note: RSA_PSK is defined in [TLSPSK].
DHE_RSA ECDHE_RSA	RSA public key; the certificate MUST allow the key to be used for signing (the digitalSignature bit MUST be set if the key usage extension is present) with the signature scheme and hash algorithm that will be employed in the server key exchange message. Note: ECDHE_RSA is defined in [TLSECC].
DHE_DSS	DSA public key; the certificate MUST allow the key to be used for signing with the hash algorithm that will be employed in the server key exchange message.
DH_DSS DH_RSA	Diffie-Hellman public key; the keyAgreement bit MUST be set if the key usage extension is present.
ECDH_ECDSA ECDH_RSA	ECDH-capable public key; the public key MUST use a curve and point format supported by the client, as described in [TLSECC].
ECDHE_ECDSA	ECDSA-capable public key; the certificate MUST allow the key to be used for signing with the hash algorithm that will be employed in the server key exchange message. The public key MUST use a curve and point format supported by the client, as described in [TLSECC].

Source: https://tools.ietf.org/html/rfc5246 at 48-49, Last accessed on Mar 19, 2020

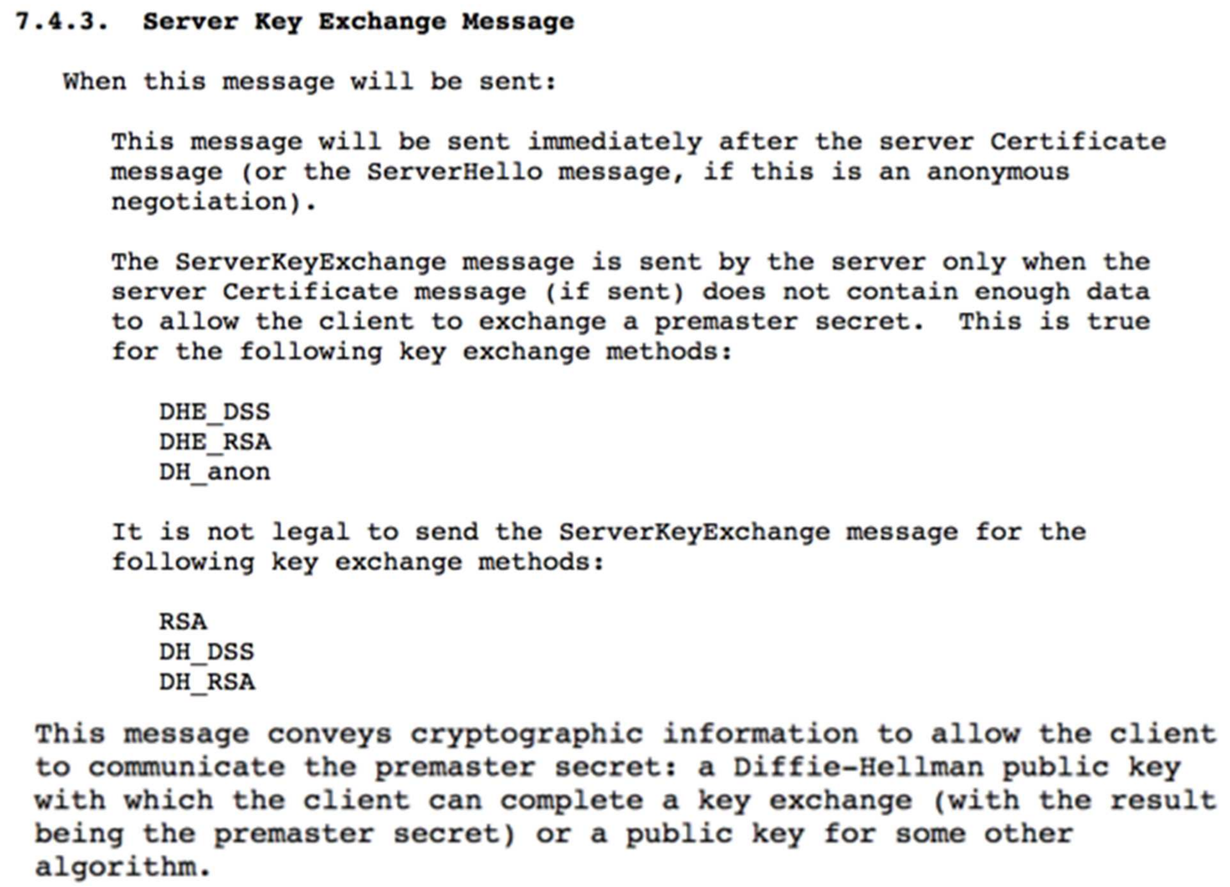
	Each of these algorithms necessitates generating one or more asymmetric key pairs – that are in turn used to compute a shared master secret for encrypting the mobile app download. For **RSA and RSA _PSK**, Apple server generates an RSA public-private key pair. For **DHE_RSA, ECDHE_RSA, DH_RSA and ECDH_RSA**, Apple server generates an RSA public-private key pair as well as a Diffie-Hellman public-private key pair[37]. The user's remote device also generates a second Diffie-Hellman public-private key pair. For **DHE_DSS and DH_DSS**, Apple server generates a DSA public-private key pair as well as a Diffie-Hellman public-private key pair. The user's remote device also generates a second Diffie-Hellman public-private key pair. For **ECDH_ECDSA and ECDHE_ECDSA**, Apple server generates an ECDSA public-private key pair as well as a Diffie-Hellman public-private key pair. The user's remote device also generates a second Diffie-Hellman public-private key pair.

[37] ECDH and ECDHE algorithms require generating at the server and the client, elliptical curve parameters that constitute a Diffie-Hellman public-private key pair. See, for example, https://tools.ietf.org/html/rfc5246 page 49-52, https://tools.ietf.org/html/rfc7525 page 12, http://www.cse.hut.fi/fi/opinnot/T-110.5241/2011/luennot-files/Network%20Security%2004%20-%20TLS.pdf, http://csrc.nist.gov/groups/STM/cmvp/documents/140-1/140sp/140sp2897.pdf, http://www.networkworld.com/article/2268575/lan-wan/chapter-2--ssl-vpn-technology.html, http://homes.esat.kuleuven.be/~fvercaut/papers/ACM2012.pdf, Implementing SSL / TLS Using Cryptography and PKI by Joshua Davies, ISBN 1118038770, 9781118038772, Page 305 and Introduction to Computer Networks and Cybersecurity By Chwan-Hwa (John) Wu, J. David Irwin, ISBN 1466572140, 9781466572140, Page 1021, which state that ECDH requires generating a Diffie-Hellman public-private key pair, Last accessed on Mar 19, 2020



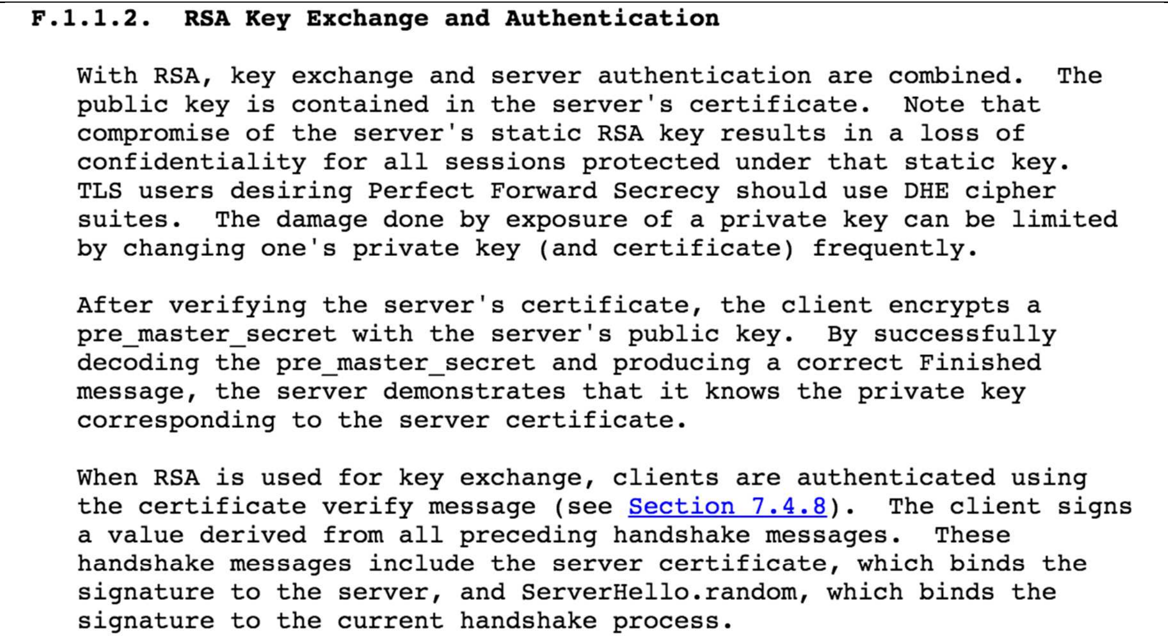
```
DHE_RSA                RSA public key; the certificate MUST allow the
ECDHE_RSA              key to be used for signing (the
                       digitalSignature bit MUST be set if the key
                       usage extension is present) with the signature
                       scheme and hash algorithm that will be employed
                       in the server key exchange message.
                       Note: ECDHE_RSA is defined in [TLSECC].

DHE_DSS                DSA public key; the certificate MUST allow the
                       key to be used for signing with the hash
                       algorithm that will be employed in the server
                       key exchange message.

DH_DSS                 Diffie-Hellman public key; the keyAgreement bit
DH_RSA                 MUST be set if the key usage extension is
                       present.

ECDH_ECDSA             ECDH-capable public key; the public key MUST
ECDH_RSA               use a curve and point format supported by the
                       client, as described in [TLSECC].

ECDHE_ECDSA            ECDSA-capable public key; the certificate MUST
                       allow the key to be used for signing with the
                       hash algorithm that will be employed in the
                       server key exchange message.  The public key
                       MUST use a curve and point format supported by
                       the client, as described in  [TLSECC].
```

Source: https://tools.ietf.org/html/rfc5246 at 48-49, Last accessed on Mar 19, 2020



```
7.4.3.  Server Key Exchange Message

   When this message will be sent:

      This message will be sent immediately after the server Certificate
      message (or the ServerHello message, if this is an anonymous
      negotiation).

      The ServerKeyExchange message is sent by the server only when the
      server Certificate message (if sent) does not contain enough data
      to allow the client to exchange a premaster secret.  This is true
      for the following key exchange methods:

         DHE_DSS
         DHE_RSA
         DH_anon

      It is not legal to send the ServerKeyExchange message for the
      following key exchange methods:

         RSA
         DH_DSS
         DH_RSA

 This message conveys cryptographic information to allow the client
 to communicate the premaster secret: a Diffie-Hellman public key
 with which the client can complete a key exchange (with the result
 being the premaster secret) or a public key for some other
 algorithm.
```

Source: https://tools.ietf.org/html/rfc5246 at 50-51, Last accessed on Mar 19, 2020

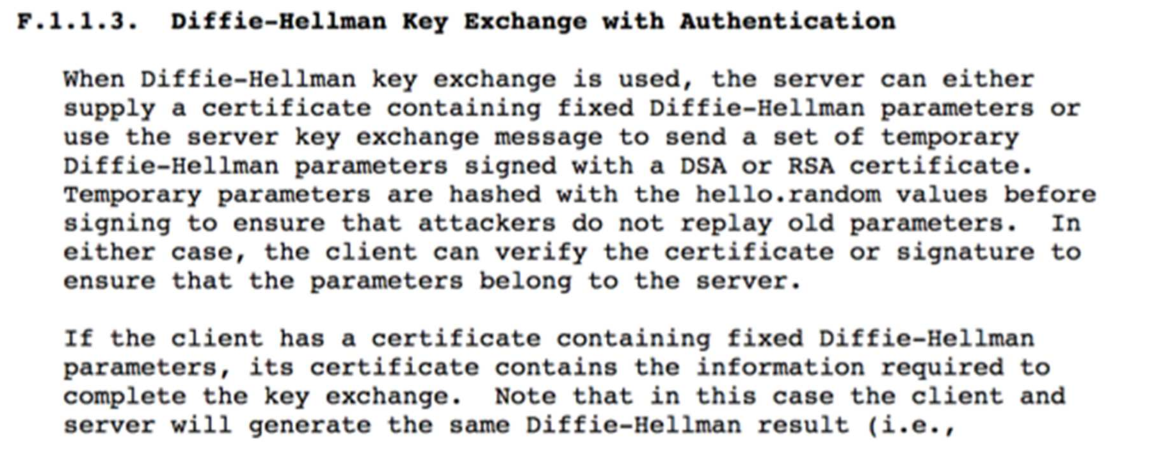


F.1.1.2. RSA Key Exchange and Authentication

With RSA, key exchange and server authentication are combined. The public key is contained in the server's certificate. Note that compromise of the server's static RSA key results in a loss of confidentiality for all sessions protected under that static key. TLS users desiring Perfect Forward Secrecy should use DHE cipher suites. The damage done by exposure of a private key can be limited by changing one's private key (and certificate) frequently.

After verifying the server's certificate, the client encrypts a pre_master_secret with the server's public key. By successfully decoding the pre_master_secret and producing a correct Finished message, the server demonstrates that it knows the private key corresponding to the server certificate.

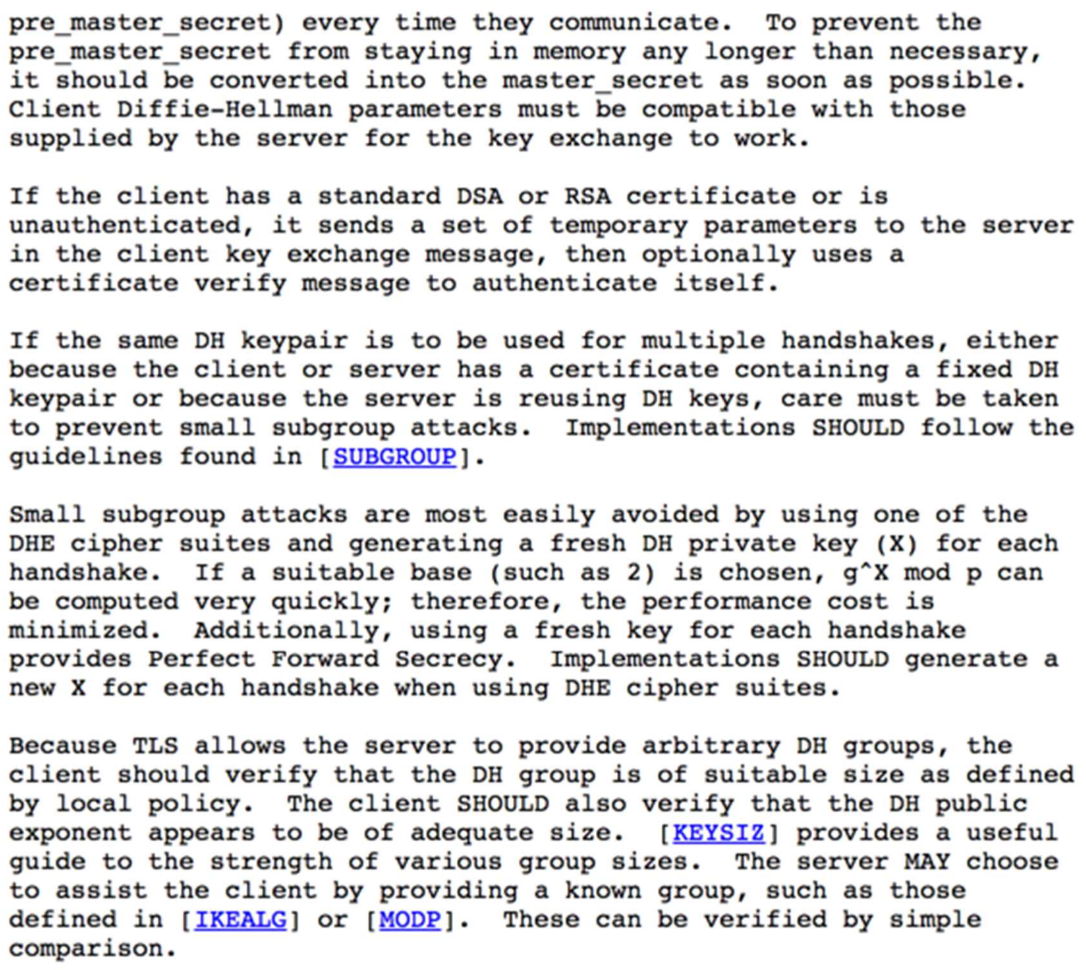
When RSA is used for key exchange, clients are authenticated using the certificate verify message (see Section 7.4.8). The client signs a value derived from all preceding handshake messages. These handshake messages include the server certificate, which binds the signature to the server, and ServerHello.random, which binds the signature to the current handshake process.



F.1.1.3. Diffie-Hellman Key Exchange with Authentication

When Diffie-Hellman key exchange is used, the server can either supply a certificate containing fixed Diffie-Hellman parameters or use the server key exchange message to send a set of temporary Diffie-Hellman parameters signed with a DSA or RSA certificate. Temporary parameters are hashed with the hello.random values before signing to ensure that attackers do not replay old parameters. In either case, the client can verify the certificate or signature to ensure that the parameters belong to the server.

If the client has a certificate containing fixed Diffie-Hellman parameters, its certificate contains the information required to complete the key exchange. Note that in this case the client and server will generate the same Diffie-Hellman result (i.e.,



```
pre_master_secret) every time they communicate.  To prevent the
pre_master_secret from staying in memory any longer than necessary,
it should be converted into the master_secret as soon as possible.
Client Diffie-Hellman parameters must be compatible with those
supplied by the server for the key exchange to work.

If the client has a standard DSA or RSA certificate or is
unauthenticated, it sends a set of temporary parameters to the server
in the client key exchange message, then optionally uses a
certificate verify message to authenticate itself.

If the same DH keypair is to be used for multiple handshakes, either
because the client or server has a certificate containing a fixed DH
keypair or because the server is reusing DH keys, care must be taken
to prevent small subgroup attacks.  Implementations SHOULD follow the
guidelines found in [SUBGROUP].

Small subgroup attacks are most easily avoided by using one of the
DHE cipher suites and generating a fresh DH private key (X) for each
handshake.  If a suitable base (such as 2) is chosen, g^X mod p can
be computed very quickly; therefore, the performance cost is
minimized.  Additionally, using a fresh key for each handshake
provides Perfect Forward Secrecy.  Implementations SHOULD generate a
new X for each handshake when using DHE cipher suites.

Because TLS allows the server to provide arbitrary DH groups, the
client should verify that the DH group is of suitable size as defined
by local policy.  The client SHOULD also verify that the DH public
exponent appears to be of adequate size.  [KEYSIZ] provides a useful
guide to the strength of various group sizes.  The server MAY choose
to assist the client by providing a known group, such as those
defined in [IKEALG] or [MODP].  These can be verified by simple
comparison.
```

Source: The Transport Layer Security (TLS) Protocol Version 1.2, IETF RFC 5246, https://tools.ietf.org/html/rfc5246, Last accessed on Mar 19, 2020

The generated asymmetric key pairs are then used to compute a shared master secret which is then used to encrypt the mobile app download so that it is resistant to observation during transit.

	For **RSA and RSA _PSK**, the RSA public-private key pair is used to encrypt a random premaster secret which is in turn used by Apple server and the user's remote device to compute a master secret. Apple uses the master secret to encrypt the mobile app and the user's remote device uses to decrypt the downloaded mobile app according to the TLS protocol. For **DHE_RSA, ECDHE_RSA, DH_RSA and ECDH_RSA**, Apple server generates an RSA public-private key pair as well as a Diffie-Hellman public-private key pair[38]. The user's remote device also generates a second Diffie-Hellman public-private key pair. Apple server uses its Diffie-Hellman private key and the user's Diffie-Hellman public key to compute a premaster secret, and subsequently compute a master secret. The user's remote device uses the user's Diffie-Hellman private key and Apple's Diffie-Hellman public key to compute the same premaster secret and subsequently the master secret that Apple uses to encrypt the mobile app and the user's remote device uses to decrypt the downloaded mobile app according to the TLS protocol. For **DHE_DSS and DH_DSS**, Apple server generates a DSA public-private key pair as well as a Diffie-Hellman public-private key pair. The user's remote device also generates a second Diffie-Hellman public-private key pair. Apple server uses its Diffie-Hellman private key and the user's Diffie-Hellman public key to compute a premaster secret, and subsequently compute a master secret. The user's remote device uses the user's Diffie-Hellman private key and Apple's Diffie-Hellman public key to compute the same premaster secret and subsequently the master secret that Apple uses to encrypt the mobile app and the user's remote device uses to decrypt the downloaded mobile app according to the TLS protocol. For **ECDH_ECDSA and ECDHE_ECDSA**, Apple server generates an ECDSA public-private key pair as well as a Diffie-Hellman public-private key pair. The user's remote device also generates a second Diffie-Hellman public-private key pair. Apple server uses its Diffie-Hellman private key and the user's Diffie-Hellman public key to compute a premaster secret, and subsequently compute a master secret. The user's remote device uses the user's Diffie-Hellman private key and Apple's Diffie-Hellman public key to compute the same premaster secret and subsequently the master secret that Apple uses to encrypt the mobile app and the user's remote device uses to decrypt the downloaded mobile app according to the TLS protocol. **3. Resistant to Modification Because Mobile App is Code Signed** The downloaded mobile app code is resistant to modification, at least in part, because the downloaded app binary is code signed. Code-signing allows users' remote systems to verify that the downloaded app binary is authentic and has not been maliciously modified by a third party. Apple

[38] ECDH and ECDHE algorithms require generating at the server and the client, elliptical curve parameters that constitute a Diffie-Hellman public-private key pair. See, for example, https://tools.ietf.org/html/rfc5246 page 49-52, https://tools.ietf.org/html/rfc7525 page 12, http://www.cse.hut.fi/fi/opinnot/T-110.5241/2011/luennot-files/Network%20Security%2004%20-%20TLS.pdf, http://csrc.nist.gov/groups/STM/cmvp/documents/140-1/140sp/140sp2897.pdf, http://www.networkworld.com/article/2268575/lan-wan/chapter-2--ssl-vpn-technology.html, http://homes.esat.kuleuven.be/~fvercaut/papers/ACM2012.pdf, Implementing SSL / TLS Using Cryptography and PKI by Joshua Davies, ISBN 1118038770, 9781118038772, Page 305 and Introduction to Computer Networks and Cybersecurity By Chwan-Hwa (John) Wu, J. David Irwin, ISBN
1466572140, 9781466572140, Page 1021, which state that ECDH requires generating a Diffie-Hellman public-private key pair, Last accessed on Mar 19, 2020

dictates that each developer must sign the mobile app submission with his/her asymmetric developer key that certifies that the app has not been modified by a third party impersonator[39].

Xcode code signs your app during the build and archive process. If needed, Xcode requests a certificate and adds a signing certificate, the certificate with its public-private key pair, to your keychain. The certificate with the public key is added to your developer account.

Source: https://help.apple.com/xcode/mac/current/#/dev3a05256b8, Last accessed on Mar 19, 2020

A signing signing certificate includes the certificate with its public-private key pair issued by Apple, and is stored in your keychain. Because the private key is stored locally, protect it as you would an account password. An intermediate certificate is also required to be in your keychain to ensure that your certificate is issued by a certificate authority such as Apple.
Your signing certificate is added to your keychain and the corresponding certificate is added to your developer account.

Source: https://help.apple.com/xcode/mac/current/#/dev1c7c2c67d, Last accessed on Mar 19, 2020

About Signing Identities and Certificates

Code signing (or signing) an app allows the system to identify who signed the app and to verify that the app has not been modified since it was signed.

Signing is a requirement for uploading your app to App Store Connect and distributing it through TestFlight or the App Store. The operating system verifies the signature of apps downloaded from the App Store to ensure that apps with invalid signatures don't run. An app's executable code is protected by its signature because the signature becomes invalid if any of the executable code in the app bundle changes. A valid signature lets users trust that the app was signed by an Apple source and hasn't been modified since it was signed.

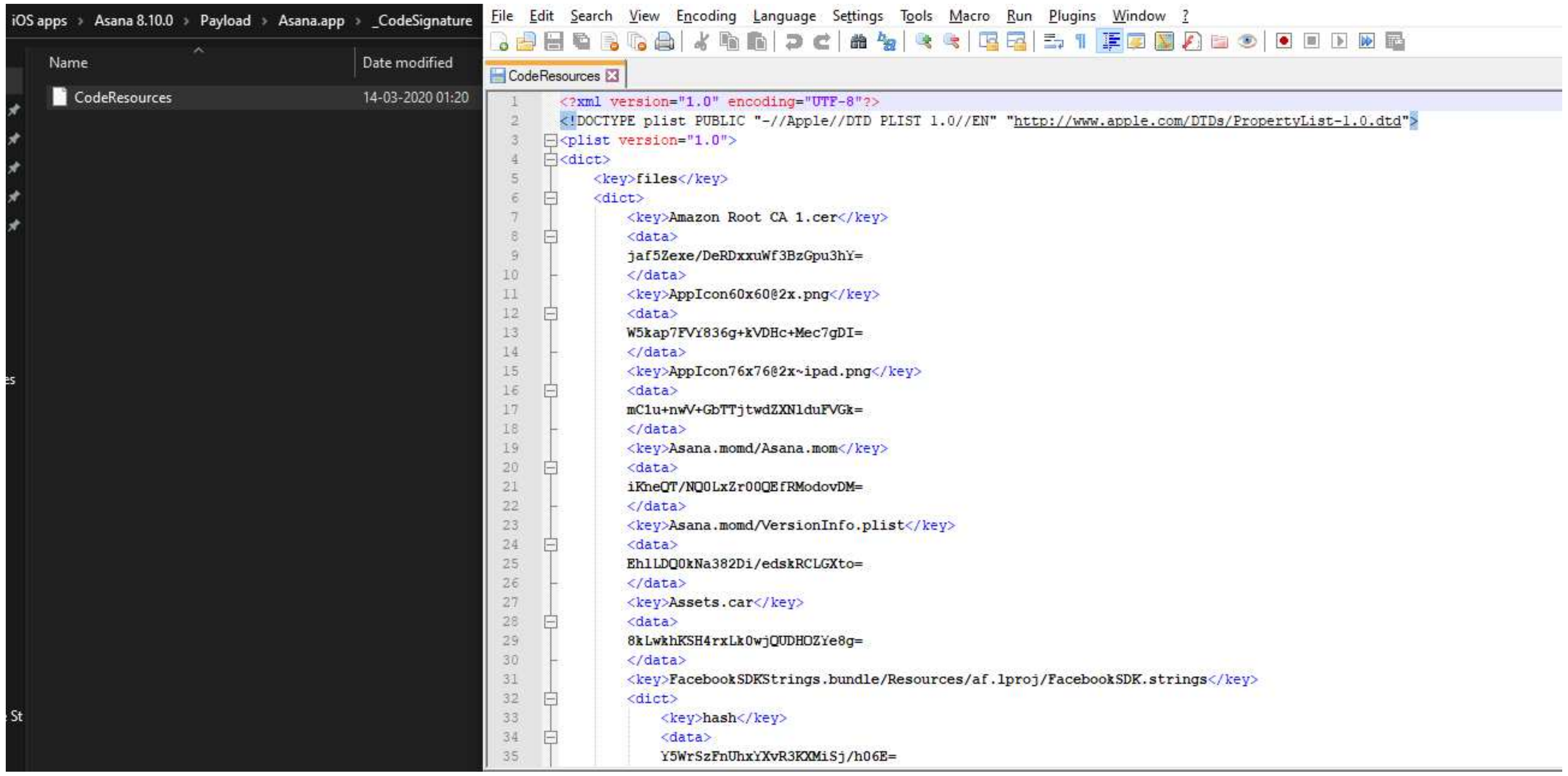
Xcode uses your signing certificate to sign your app during the build process. The signing certificate consists of a public-private key pair and a certificate. The private key is used by cryptographic functions to generate the signature. The certificate is issued by Apple; it contains the public key and identifies you as the owner of the key pair. In order to sign apps, you must have both parts of your signing certificate, and an Apple certificate authority in your keychain.

[39] *See, e.g.*, https://developer.apple.com/library/ios/documentation/IDEs/Conceptual/AppDistributionGuide/MaintainingCertificates/MaintainingCertificates.html, Last accessed on Mar 19, 2020

An app's signature can be removed, and the app can be re-signed using another signing certificate. For example, Apple re-signs all apps sold on the App Store. Also, a fully-tested development build of your app can be re-signed for submission to the App Store. Thus the signature is best understood not as proof of the app's origin but as a verifiable mark placed by the signer.

Source: https://help.apple.com/xcode/mac/current/#/devfbe995ebf, Last accessed on Mar 19, 2020

Asana complies with Apple's instructions on code signing as shown by Asana's mobile app contents. Asana's mobile apps contain files such as the file _CodeSignature/CodeResources in Asana's iOS apps which are generated during the code signing process as per instructions from Apple.



iOS apps › Asana 8.10.0 › Payload › Asana.app › _CodeSignature

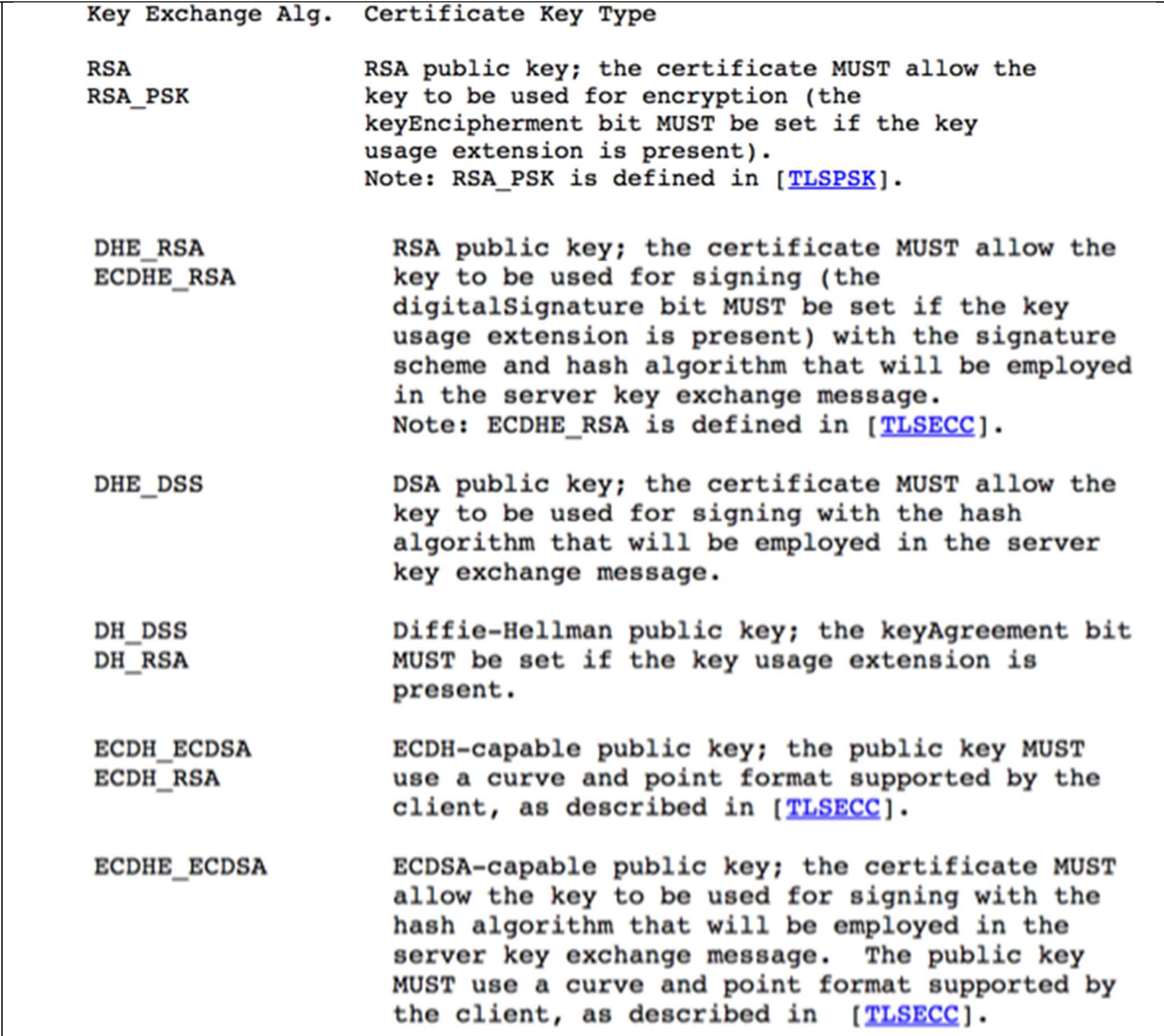
Name	Date modified
CodeResources	14-03-2020 01:20

CodeResources

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN" "http://www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
<dict>
    <key>files</key>
    <dict>
        <key>Amazon Root CA 1.cer</key>
        <data>
        jaf5Zexe/DeRDxxuWf3BzGpu3hY=
        </data>
        <key>AppIcon60x60@2x.png</key>
        <data>
        W5kap7FVY836g+kVDHc+Mec7gDI=
        </data>
        <key>AppIcon76x76@2x~ipad.png</key>
        <data>
        mC1u+nwV+GbTTjtwdZXNlduFVGk=
        </data>
        <key>Asana.momd/Asana.mom</key>
        <data>
        iKneQT/NQ0LxZr00QEfRModovDM=
        </data>
        <key>Asana.momd/VersionInfo.plist</key>
        <data>
        EhlLDQ0kNa382Di/edskRCLGXto=
        </data>
        <key>Assets.car</key>
        <data>
        8kLwkhKSH4rxLk0wjQUDHOZYe8g=
        </data>
        <key>FacebookSDKStrings.bundle/Resources/af.lproj/FacebookSDK.strings</key>
        <dict>
            <key>hash</key>
            <data>
            Y5WrSzFnUhxYXvR3KXMiSj/h06E=
```

	Source: Contents of Asana: organize tasks & work, https://apps.apple.com/us/app/asana-organize-tasks-work/id489969512, as an example of Asana app, Last accessed on Mar 19, 2020 By signing the app binary with a digital signature, Asana's mobile apps are tamper resistant enabling Apple and the iOS mobile devices to verify that the application is being distributed by trusted source (*i.e.* Asana) and that the application has not been modified by a third party, which can be verified by the corresponding public key generated as part of the pair. Thus the app binary is made resistant to modification by digital signing. Accordingly Asana's iOS mobile apps establish SSL/TLS communications with Asana's servers, which involve a SSL/TLS handshake procedure involving asymmetric key encryption. SSL/TLS ensures secure communication and renders the mobile app data further resistant to observation. **4. Resistant to Observation Because Mobile App is Stored on Remote System in Encrypted Form** The mobile app is made further resistant to observation because when downloaded and installed on a user's iOS mobile device, it is stored in an encrypted form. iOS implements disk encryption for encrypting the operating system software, apps and all related data on a mobile device – which further renders Asana app resistant to observation[40]. **5. Resistant to Observation Because Mobile App Securely Communicates with Asana Over SSL/TLS** Asana's mobile apps are made further resistant to observation, at least in part, because the mobile app communicates with Asana using SSL/TLS during operation. Asana app users establish SSL/TLS communications with Asana servers when the app is executed. Such secure communication is necessary to keep source code as well as user identity and activity from being observed in transit from the remote system to Asana servers and vice versa. The secure communications process starts with a TLS handshake procedure which uses asymmetric key encryption and necessitates generation of at least one asymmetric key pair. Specifically, user's remote device negotiates with Asana servers the cipher suite and the key exchange algorithm that will be used for the handshake.

[40] *See, e.g.*, https://www.apple.com/business/docs/iOS_Security_Guide.pdf, Pages 10-18*,* Last accessed on May 22, 2019

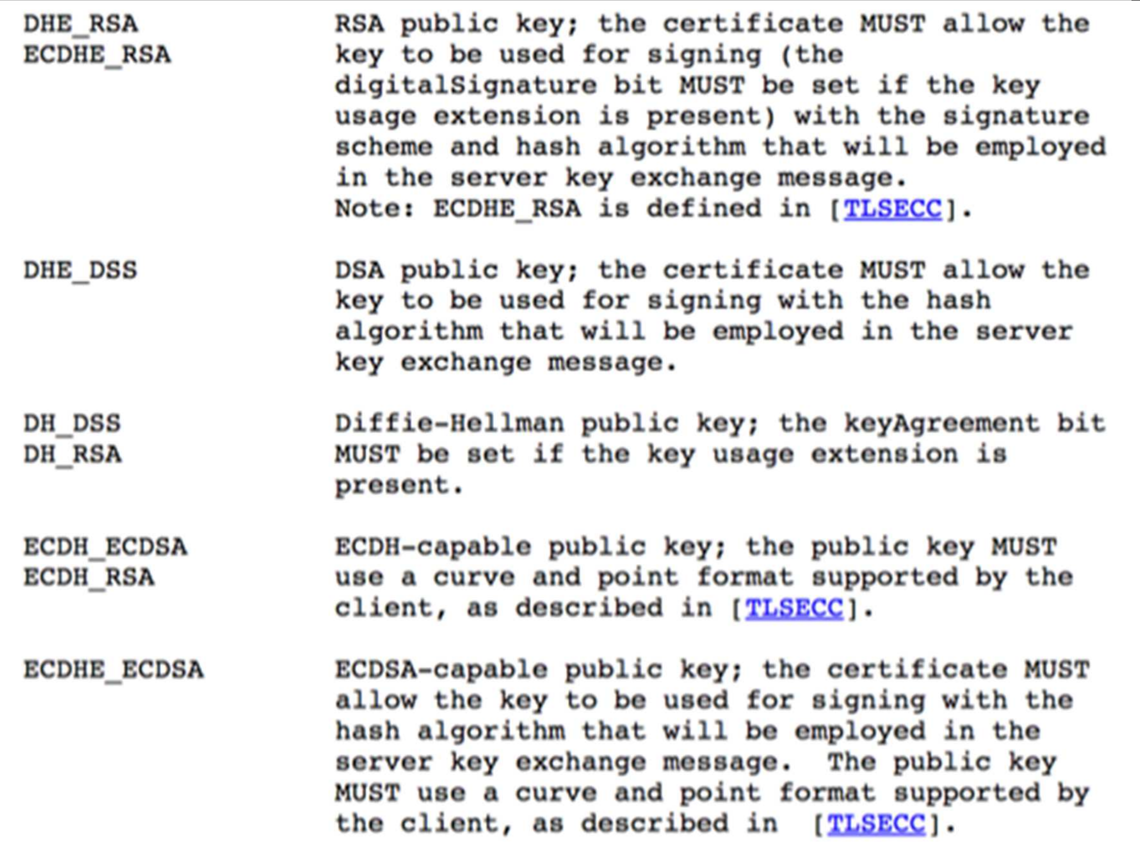


```
Key Exchange Alg.  Certificate Key Type

RSA                RSA public key; the certificate MUST allow the
RSA_PSK            key to be used for encryption (the
                   keyEncipherment bit MUST be set if the key
                   usage extension is present).
                   Note: RSA_PSK is defined in [TLSPSK].

DHE_RSA            RSA public key; the certificate MUST allow the
ECDHE_RSA          key to be used for signing (the
                   digitalSignature bit MUST be set if the key
                   usage extension is present) with the signature
                   scheme and hash algorithm that will be employed
                   in the server key exchange message.
                   Note: ECDHE_RSA is defined in [TLSECC].

DHE_DSS            DSA public key; the certificate MUST allow the
                   key to be used for signing with the hash
                   algorithm that will be employed in the server
                   key exchange message.

DH_DSS             Diffie-Hellman public key; the keyAgreement bit
DH_RSA             MUST be set if the key usage extension is
                   present.

ECDH_ECDSA         ECDH-capable public key; the public key MUST
ECDH_RSA           use a curve and point format supported by the
                   client, as described in [TLSECC].

ECDHE_ECDSA        ECDSA-capable public key; the certificate MUST
                   allow the key to be used for signing with the
                   hash algorithm that will be employed in the
                   server key exchange message.  The public key
                   MUST use a curve and point format supported by
                   the client, as described in  [TLSECC].
```

Source: https://tools.ietf.org/html/rfc5246 at 48-49, Last accessed on Mar 19, 2020

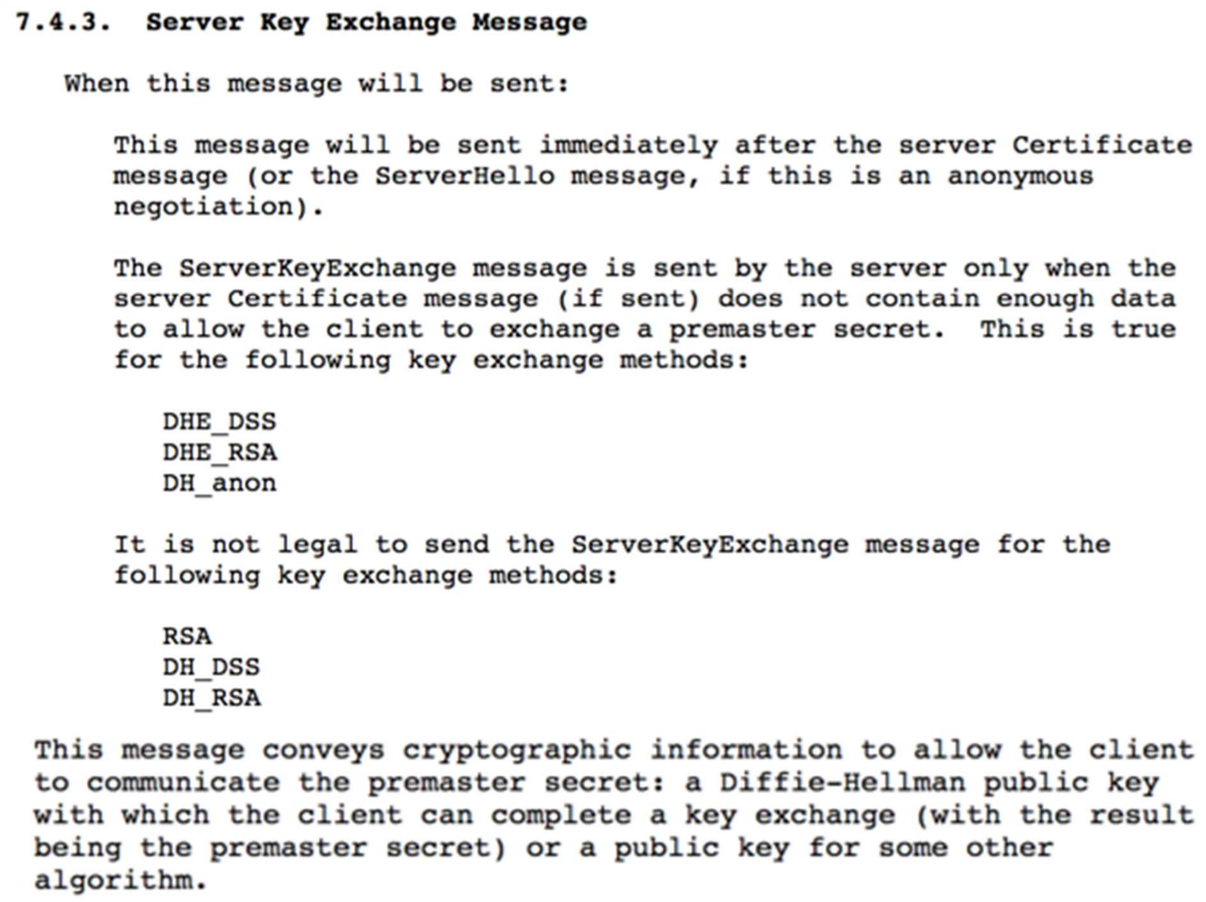
	Each of these algorithms necessitates generating one or more asymmetric key pairs – that are in turn used to compute a shared master secret for encrypting communication between Asana and user's remote device. For **RSA and RSA _PSK**, Asana server generates an RSA public-private key pair. For **DHE_RSA, ECDHE_RSA, DH_RSA and ECDH_RSA**, Asana server generates an RSA public-private key pair as well as a Diffie-Hellman public-private key pair[41]. The user's remote device also generates a second Diffie-Hellman public-private key pair. For **DHE_DSS and DH_DSS**, Asana server generates a DSA public-private key pair as well as a Diffie-Hellman public-private key pair. The user's remote device also generates a second Diffie-Hellman public-private key pair. For **ECDH_ECDSA and ECDHE_ECDSA**, Asana server generates an ECDSA public-private key pair as well as a Diffie-Hellman public-private key pair. The user's remote device also generates a second Diffie-Hellman public-private key pair.

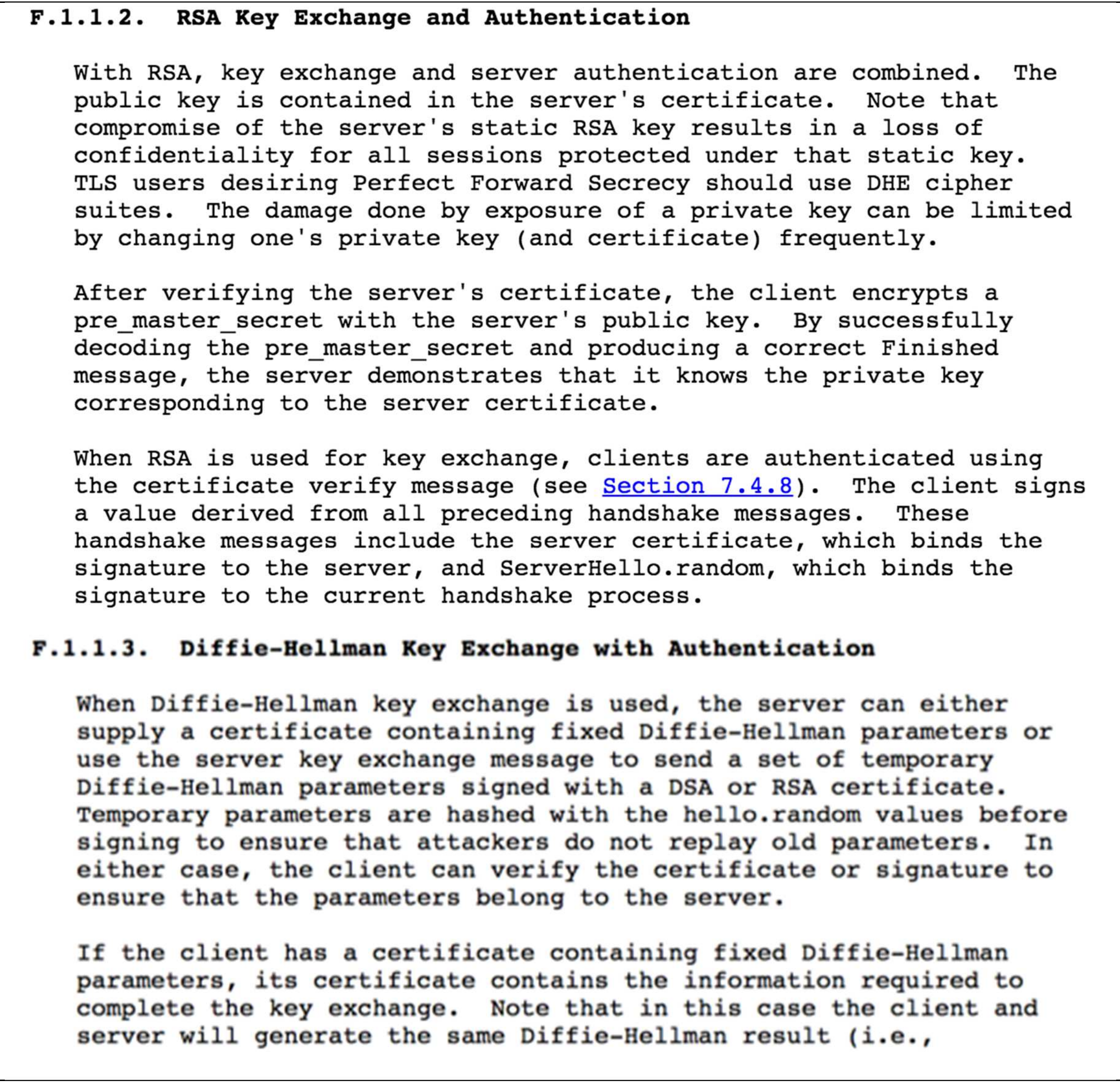
[41] ECDH and ECDHE algorithms require generating at the server and the client, elliptical curve parameters that constitute a Diffie-Hellman public-private key pair. See, for example, https://tools.ietf.org/html/rfc5246 page 49-52, https://tools.ietf.org/html/rfc7525 page 12, http://www.cse.hut.fi/fi/opinnot/T-110.5241/2011/luennot-files/Network%20Security%2004%20-%20TLS.pdf, http://csrc.nist.gov/groups/STM/cmvp/documents/140-1/140sp/140sp2897.pdf, http://www.networkworld.com/article/2268575/lan-wan/chapter-2--ssl-vpn-technology.html, http://homes.esat.kuleuven.be/~fvercaut/papers/ACM2012.pdf, Implementing SSL / TLS Using Cryptography and PKI by Joshua Davies, ISBN 1118038770, 9781118038772, Page 305 and Introduction to Computer Networks and Cybersecurity By Chwan-Hwa (John) Wu, J. David Irwin, ISBN 1466572140, 9781466572140, Page 1021, which state that ECDH requires generating a Diffie-Hellman public-private key pair, Last accessed on Mar 19, 2020



```
   DHE_RSA              RSA public key; the certificate MUST allow the
   ECDHE_RSA            key to be used for signing (the
                        digitalSignature bit MUST be set if the key
                        usage extension is present) with the signature
                        scheme and hash algorithm that will be employed
                        in the server key exchange message.
                        Note: ECDHE_RSA is defined in [TLSECC].

   DHE_DSS              DSA public key; the certificate MUST allow the
                        key to be used for signing with the hash
                        algorithm that will be employed in the server
                        key exchange message.

   DH_DSS               Diffie-Hellman public key; the keyAgreement bit
   DH_RSA               MUST be set if the key usage extension is
                        present.

   ECDH_ECDSA           ECDH-capable public key; the public key MUST
   ECDH_RSA             use a curve and point format supported by the
                        client, as described in [TLSECC].

   ECDHE_ECDSA          ECDSA-capable public key; the certificate MUST
                        allow the key to be used for signing with the
                        hash algorithm that will be employed in the
                        server key exchange message.  The public key
                        MUST use a curve and point format supported by
                        the client, as described in  [TLSECC].
```

Source: https://tools.ietf.org/html/rfc5246 at 48-49, Last accessed on Mar 19, 2020

	 **7.4.3. Server Key Exchange Message** When this message will be sent: This message will be sent immediately after the server Certificate message (or the ServerHello message, if this is an anonymous negotiation). The ServerKeyExchange message is sent by the server only when the server Certificate message (if sent) does not contain enough data to allow the client to exchange a premaster secret. This is true for the following key exchange methods: DHE_DSS DHE_RSA DH_anon It is not legal to send the ServerKeyExchange message for the following key exchange methods: RSA DH_DSS DH_RSA This message conveys cryptographic information to allow the client to communicate the premaster secret: a Diffie-Hellman public key with which the client can complete a key exchange (with the result being the premaster secret) or a public key for some other algorithm. *Source: https://tools.ietf.org/html/rfc5246 at 50-51,* Last accessed on Mar 19, 2020



F.1.1.2. RSA Key Exchange and Authentication

With RSA, key exchange and server authentication are combined. The public key is contained in the server's certificate. Note that compromise of the server's static RSA key results in a loss of confidentiality for all sessions protected under that static key. TLS users desiring Perfect Forward Secrecy should use DHE cipher suites. The damage done by exposure of a private key can be limited by changing one's private key (and certificate) frequently.

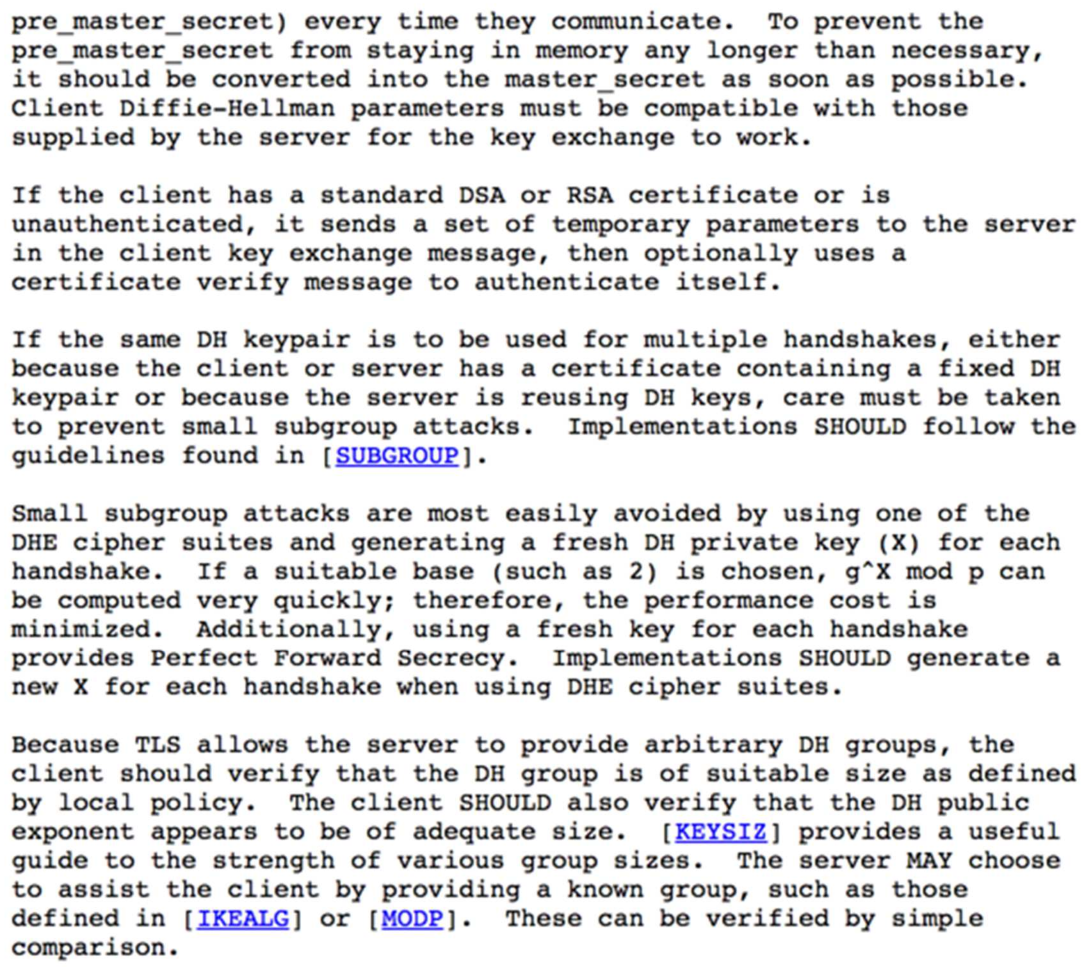
After verifying the server's certificate, the client encrypts a pre_master_secret with the server's public key. By successfully decoding the pre_master_secret and producing a correct Finished message, the server demonstrates that it knows the private key corresponding to the server certificate.

When RSA is used for key exchange, clients are authenticated using the certificate verify message (see Section 7.4.8). The client signs a value derived from all preceding handshake messages. These handshake messages include the server certificate, which binds the signature to the server, and ServerHello.random, which binds the signature to the current handshake process.

F.1.1.3. Diffie-Hellman Key Exchange with Authentication

When Diffie-Hellman key exchange is used, the server can either supply a certificate containing fixed Diffie-Hellman parameters or use the server key exchange message to send a set of temporary Diffie-Hellman parameters signed with a DSA or RSA certificate. Temporary parameters are hashed with the hello.random values before signing to ensure that attackers do not replay old parameters. In either case, the client can verify the certificate or signature to ensure that the parameters belong to the server.

If the client has a certificate containing fixed Diffie-Hellman parameters, its certificate contains the information required to complete the key exchange. Note that in this case the client and server will generate the same Diffie-Hellman result (i.e.,



pre_master_secret) every time they communicate. To prevent the pre_master_secret from staying in memory any longer than necessary, it should be converted into the master_secret as soon as possible. Client Diffie-Hellman parameters must be compatible with those supplied by the server for the key exchange to work.

If the client has a standard DSA or RSA certificate or is unauthenticated, it sends a set of temporary parameters to the server in the client key exchange message, then optionally uses a certificate verify message to authenticate itself.

If the same DH keypair is to be used for multiple handshakes, either because the client or server has a certificate containing a fixed DH keypair or because the server is reusing DH keys, care must be taken to prevent small subgroup attacks. Implementations SHOULD follow the guidelines found in [SUBGROUP].

Small subgroup attacks are most easily avoided by using one of the DHE cipher suites and generating a fresh DH private key (X) for each handshake. If a suitable base (such as 2) is chosen, g^X mod p can be computed very quickly; therefore, the performance cost is minimized. Additionally, using a fresh key for each handshake provides Perfect Forward Secrecy. Implementations SHOULD generate a new X for each handshake when using DHE cipher suites.

Because TLS allows the server to provide arbitrary DH groups, the client should verify that the DH group is of suitable size as defined by local policy. The client SHOULD also verify that the DH public exponent appears to be of adequate size. [KEYSIZ] provides a useful guide to the strength of various group sizes. The server MAY choose to assist the client by providing a known group, such as those defined in [IKEALG] or [MODP]. These can be verified by simple comparison.

Source: The Transport Layer Security (TLS) Protocol Version 1.2, IETF RFC 5246, https://tools.ietf.org/html/rfc5246, Last accessed on Mar 19, 2020

The generated asymmetric key pairs are then used to compute a shared master secret which is then used to encrypt subsequent communications between Asana and the user’s remote device so that they are resistant to observation during transit.

	For **RSA and RSA _PSK**, the RSA public-private key pair is used to encrypt a random premaster secret which is in turn used by Asana server and the user's remote device to compute a master secret. Asana and the user's remote device use the master secret for encrypting and decrypting communication messages. For **DHE_RSA, ECDHE_RSA, DH_RSA and ECDH_RSA**, Asana server generates an RSA public-private key pair as well as a Diffie-Hellman public-private key pair[42]. The user's remote device also generates a second Diffie-Hellman public-private key pair. Asana server uses its Diffie-Hellman private key and the user's Diffie-Hellman public key to compute a premaster secret, and subsequently compute a master secret. The user's remote device uses the user's Diffie-Hellman private key and Apple's Diffie-Hellman public key to compute the same premaster secret and subsequently the master secret for encrypting and decrypting communication messages. For **DHE_DSS and DH_DSS**, Asana server generates a DSA public-private key pair as well as a Diffie-Hellman public-private key pair. The user's remote device also generates a second Diffie-Hellman public-private key pair. Asana server uses its Diffie-Hellman private key and the user's Diffie-Hellman public key to compute a premaster secret, and subsequently compute a master secret. The user's remote device uses the user's Diffie-Hellman private key and Apple's Diffie-Hellman public key to compute the same premaster secret and subsequently the master secret for encrypting and decrypting communication messages. For **ECDH_ECDSA and ECDHE_ECDSA**, Asana server generates an ECDSA public-private key pair as well as a Diffie-Hellman public-private key pair. The user's remote device also generates a second Diffie-Hellman public-private key pair. Asana server uses its Diffie-Hellman private key and the user's Diffie-Hellman public key to compute a premaster secret, and subsequently compute a master secret. The user's remote device uses the user's Diffie-Hellman private key and Apple's Diffie-Hellman public key to compute the same premaster secret and subsequently the master secret for encrypting and decrypting communication messages.

[42] ECDH and ECDHE algorithms require generating at the server and the client, elliptical curve parameters that constitute a Diffie-Hellman public-private key pair. See, for example, https://tools.ietf.org/html/rfc5246 page 49-52, https://tools.ietf.org/html/rfc7525 page 12, http://www.cse.hut.fi/fi/opinnot/T-110.5241/2011/luennot-files/Network%20Security%2004%20-%20TLS.pdf, http://csrc.nist.gov/groups/STM/cmvp/documents/140-1/140sp/140sp2897.pdf, http://www.networkworld.com/article/2268575/lan-wan/chapter-2--ssl-vpn-technology.html, http://homes.esat.kuleuven.be/~fvercaut/papers/ACM2012.pdf, Implementing SSL / TLS Using Cryptography and PKI by Joshua Davies, ISBN 1118038770, 9781118038772, Page 305 and Introduction to Computer Networks and Cybersecurity By Chwan-Hwa (John) Wu, J. David Irwin, ISBN
1466572140, 9781466572140, Page 1021, which state that ECDH requires generating a Diffie-Hellman public-private key pair, Last accessed on Mar 19, 2020